MEETING DATA SHARING NEEDS OF HETEROGENEOUS DISTRIBUTED USERS

A Thesis
Presented to
The Academic Faculty

by

Zhiyuan Zhan

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
May 2007

MEETING DATA SHARING NEEDS OF HETEROGENEOUS DISTRIBUTED USERS

Approved by:

Professor Mustaque Ahamad, Advisor,
Committee Chair
College of Computing
Georgia Institute of Technology

Professor David E. Bakken
School of EECS
Washington State University

Professor Douglas M. Blough
School of ECE
Georgia Institute of Technology

Professor Ling Liu
College of Computing
Georgia Institute of Technology

Professor Karsten Schwan
College of Computing
Georgia Institute of Technology

Date Approved: 10 January 2007

To my parents, wife, and daughter

for their love and support.

ACKNOWLEDGEMENTS

First of all, I want to express my special thanks to my advisor, Prof. Mustaque Ahamad, for his continuous guidance, encouragement and support through so many years. He provided me with great flexibility on my research topic and carefully guided me with dedication and patience. His vision, wisdom, and knowledge have greatly impacted my understanding of the distributed computing field. He also gave me invaluable advice on many issues outside research topics, which I will forever cherish.

I would also like to thank my dissertation committee members, Prof. Doug Blough, Dave Bakken, Ling Liu, and Karsten Schwan, for many valuable suggestions and inputs. Their deep insights and broad knowledge have greatly influenced and improved my research over the years. Prof. Calton Pu also gave me valuable suggestions during thesis proposal. Their advice has greatly improved this thesis.

Prof. Michel Raynal provided invaluable feedback and suggestions when I worked on the ICDCS 2005 paper, which is a key part of this thesis. Dr. Greg Eisenhauer helped me to understand the Echo platform. Dr. Matthew Wolf helped me setting up the test environment. Dr. Brad Topol and Dr. Kiran Achyutuni provided me with great opportunities outside school to work on real world problems which have greatly broadened my vision. I really enjoyed many interesting discussions I had with Dr. Ernesto Jimenez. His passion for research has deeply impacted me. Without the help of people acknowledged here, I could not imagine finishing up the degree.

I am also thankful for being a member of the College of Computing family, where I received inspiration, encouragement and help from many faculty members outside my committee: Prof. Wenke Lee, Leo Mark, Ed Omiecinski, Jim Xu, and H. Venkateswaran, and staff members: Babara Binder, Becky Wilson, Jennifer Chisholm, Susie McClain, Deborah Mitchell and Mary Claire Thompson.

Many CoC friends have helped me in various ways over the years: Michael Covington, Lei Kong, Seung Jun, Tianying Chang, Sandip Agarwala, Jianjun Zhang, Li Xiong, Donghua Xu, and Yarong Tang. Many of them have already left Georgia Tech and I wish them the best of luck in their new endeavours. Jie Yang, Zhiliang Fan, Guangfan Zhang, Yingchuan Zhang, Ning Chen, Ke Wang and many friends from Georgia Tech Chinese Frienship Association (GTCFA) and its drama group (Yu Li, Xinjuan Bi, Dong Yang, Ying Zhu, Tao Xie, el. al.) have made my life ourside research interesting and fun. I will cherish their frienship forever.

Special thanks to my parents, Huaichen Zhan and Shuxiang Shen. They always have faith in me, which gives me the ultimate strength to go through this long journey. And to my little brother, Zhiwei Zhan, who can always cheer me up by his courage, unique sense of humor and optimism.

Finally and most importantly, I have to thank my wife, Weiyun Huang. We met each other 14 years ago. Since then, we have become classmates, best friends, partners, husband and wife, and recently parents along the way. I cannot express how much I am thankful for her love, support, patience and understanding over so many years. Without her, my life will be totally different. And my special thanks to our little angel, Chloe Junyi Zhan, who has brought incredible joy to us ever since she was born. Her smile brightens even the darkest time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The fast growth of wireless networking and mobile computing devices has enabled us to access information from anywhere at any time. However, varying user needs and system resource constraints are two major heterogeneity factors that pose a serious challenge to information sharing systems. For instance, when a new information item is produced, different users may have different requirements for when the new value should become visible. The resources that each device can contribute to such information sharing applications also vary, i.e. the resource rich nodes can afford expensive protocol to ensure high quality of shared information but others may not be able to do it. Therefore, how to enable information sharing across computing platforms with varying resources to meet different user demands is an interesting and important problem for distributed systems research.

In this thesis, we address the heterogeneity challenge faced by such information sharing systems. We assume that shared information is encapsulated in distributed objects, and we use object replication to increase the scalability and robustness of information sharing system, which introduces the consistency problem. Many consistency models have been proposed in recent years but they are either too strong and do not scale very well, or too weak to meet many users' requirements. We believe that to have a single level of consistency (as is commonly done in existing systems) cannot meet the challenge of maintaining replica consistency in a heterogeneous environment. Instead, we propose a Mixed Consistency (MC) model as a solution. We introduce an access constraints based approach to combine both strong and weak consistency models together. As a result, our model allows both strong and weak replicas of the same object to coexist in the system at the same time. In order to utilize the existing protocols of both strong and weak consistency models to implement the new MC model, we propose a MC protocol that combines existing implementations together with minimum modifications. We also introduce a non-responsive process detector

in order to tolerate crash failures and slow processes/communication links in the system. We also explore how the heterogeneity challenge can be addressed in the transportation layer by developing an agile dissemination protocol that can take into consideration both user needs and resource constraints when disseminating updates. We implement our MC protocol on top of a distributed publisher-subscriber middleware, Echo. We finally measure the performance of our MC implementation by conducting a series of experiments designed to expose the impacts of different system parameters. The results of the experiments are consistent with our expectations. Based on the functionality and performance of mixed consistency protocols, we believe that this model is effective in addressing the heterogeneity of user requirements and available resources in distributed systems.

CHAPTER I

INTRODUCTION

Billions of computers are now connected by the Internet. This has changed the nature of computing forever by providing the possibility of performing computation among a large group of autonomous computers, some of which may be thousands of miles away from each other. Those computers can not only share resources together but also work together to solve a problem. A distributed system is designed to utilize the resources of each connected computer to serve a common goal. It can deliver computing power at a much lower cost than any equivalent centralized computer. Further, it scales better because the size of the system can easily grow with demand. A distributed system can also provide better fault tolerance, since one computer's functionality can be replicated by others so its failure does not interrupt the computation. Distributed computer systems also better match the needs of those computing problems that manipulate data that is available at geographically distributed nodes.

The number of computers connected to the Internet continues to grow at a fast pace and broadband connection has become common in many households and workplaces. Distributed computing has never become so pervasive in people's daily lives. Nowadays, we rely on computers and Internet to trade stocks, book hotel and flights and even buy Christmas gifts. Important information like our medical and financial records have been maintained and shared among the computers in hospitals and banks. Homeland security and national defense also rely on information sharing in order to better prepare for the next crisis. In fact, our life depends on when and where to access certain information and we cannot imagine life without it.

With the fast spreading of mobile devices and wireless networks, new kinds of computing devices with varying computing power and communication capabilities have been connected to the Internet, besides servers and desktops. All together they can provide us the ability

to access information anywhere and any time. For the same piece of information, different users may have different requirements for how it should be shared. To be able to share information across computing platforms with varying resources and to meet different user demands is an interesting and challenging problem for distributed systems.

1.1 Challenge: Heterogeneity in Distributed Data Sharing

The rapid proliferation of mobile devices and wireless networks has made it possible to access and share information across different platforms with varying degree of computing power and network resources. For example, various cars can share traffic information through on-board communication devices. This scenario requires that the information be shared, disseminated and updated at a potentially large number of users. There are two sources of heterogeneity in such an environment: user needs and system resources. For instance, some users may not care about each new update, while others do. Also, some users can use relatively powerful systems to access the information, while others may only be able to use wireless handheld device to do so. The lack of computing power and/or network resources will prevent them from employing expensive protocols to ensure high quality of shared information at all time. Both sources suggest that the information sharing system should consider heterogeneity as a primary concern if it is to meet the needs of varied users in a wide area network.

1.2 Proposed Solution: A Mixed Consistency (MC) Model

We assume that shared information is encapsulated in distributed objects. Object replication is a common technique to increase the scalability and robustness of distributed systems ([26], [37], [41], [81], [64], [75], [48], [67], [14], [82]). It also introduces the consistency problem. Many consistency models have been proposed in recent years. The tradeoff between performance and the level of consistency presents a dilemma. For example, strong consistency such as sequential consistency ensures an unique order of operations across different replicas, but its implementation is costly and does not scale very well; the ordering guarantees provided by different weak consistency models may result in conflicting views of

shared critical information but they are relatively easy to implement with good performance. Therefore, only supporting a single level of consistency, which is commonly done in existing systems, either does not scale well or is insufficient for meeting user requirements. In order to maintain consistency in a heterogeneous environment, we are exploring an approach of employing *mixed consistency* [80] for objects which addresses both user needs and system resource constraints.

- For a particular piece of information, users can choose the consistency level they desire based on the current resource level they have. Users at resource rich nodes can access information with stronger consistency, while others may be forced to access information with weaker consistency guarantees due to lack of resources.

- Strong consistency for critical information is necessary but has a relatively high maintenance cost. In many applications, weaker consistency provides a relatively cheap yet scalable way for a large number of users to share information that has less stringent consistency requirements.

With a mixed consistency model, applications can make use of both high (strong) and low (weak) levels of consistency to meet their needs based on the availability of resources at various nodes. However, mixing leads to a new problem: how to meaningfully define consistency when resource poor nodes want to access an object replica in a weak mode whereas others want to maintain strong consistency for their replicas of this same object. Operations observed by strong replicas should appear to be executed on a single copy serially, but nodes with weak replicas do not have the resources necessary to ensure such strong ordering. Since update operations produce new values for replicas and strong replicas must observe new values in a consistent order, we develop constraints that prohibit values of updates done at weak replicas to be disseminated to strong replicas in our model. At the same time, a weak replica can observe values of updates done at strong replicas as well as at other weak replicas. Such a model can meet the needs of many applications. In order to bound the difference between weak and strong replicas, weak replicas can periodically update themselves with values written by nodes that maintain strong replicas.

The dissemination flow constraints allow us to precisely characterize the mixed consistency model, and they can easily be incorporated into a protocol that implements this model in a heterogeneous environment.

Although the mixed consistency model we propose is not limited to any particular consistency models, sequential consistency (SC) and causal consistency (CC) are the strong and weak consistency models we consider in this thesis. They both have been extensively studied over the past years and many protocols have been proposed to implement them.

1.3 Flexible Failure Handling

Crash failures are common in distributed systems. It is easier to deal with failures in a synchronous system since a timeout can indicate that the other process has failed (assuming only process crash failure is possible). It is challenging to deal with failures in an asynchronous system because we cannot distinguish a crashed process from a slow one [36]. Furthermore, to keep a slow process in the system not only slows down the system performance but also can cause other problems in the future (see Chapter 4). It is important for the mixed consistency model to be able to tolerate both crashed and slow processes. We introduce a new concept called "responsiveness" to deal with failures. We design and implement a non-responsive process detector. Using this detector, our mixed consistency model can support a very flexible downgrading/upgrading mechanism to handle non-responsive processes and/or changes of available system resources. For example, if a crashed process prevents other processes from maintaining strong consistency, other processes can downgrade to a lower level of consistency and continue to make progress. When the crashed process has recovered, others can upgrade themselves to a high level of consistency. The mechanism helps to achieve high overall system availability.

In this thesis, we also address the heterogeneity factors faced by the transport layer. We design and implement an agile dissemination protocol that combines different dissemination techniques and can adapt to different application needs and system resources. It can be used to support different consistency protocols.

1.4 Dissertation Organization

The rest of the thesis is organized into the following chapters:

Chapter 2 discusses related work. Chapter 3 introduces the mixed consistency model. It gives the formal definition of mixed consistency and illustrates how it is different from either strong or weak models. We also provides the protocol that implements the mixed consistency model. For simplicity, we assume a failure-free environment when introducing the model and protocols. Failures and their handling will be addressed later in other chapters.

Chapter 4 introduces our failure model, and discusses our approach for handling failures by introducing a new concept of responsiveness. It defines a non-responsive process detector and discusses how to build a non-responsive tolerant causal consistency (nrtcc) protocol by giving two different implementations.

Chapter 5 continues the failure handling discussion and extends the mixed consistency protocol to tolerate non-responsive processes. It is built on top of the nrtcc protocol introduced in previous chapter. Chapter 4 and Chapter 5 together completes our discussion of how the mixed consistency model can be extended to handle failures.

Chapter 6 introduces our agile communication layer, which can take into consideration different user needs by observing various read frequencies and use different dissemination techniques to meet different needs in a cost-effective way. We also describe the communication protocols that support the implementation in our consistency protocols in this chapter.

Chapter 7 describes a system that implements our mixed consistency protocol. We present the system architecture and then discuss the performance of the implementation. The performance data are obtained through a series of experiments targeted to capture different combinations of performance impacting parameters. We finally conclude the thesis and discuss future research directions in Chapter 8.

CHAPTER II

RELATED WORK

We discuss related work in this chapter. We first introduce key consistency models that are closely related to our work, namely sequential consistency (SC) and causal consistency (CC), and briefly survey some other known models that have been developed in previous work. We then discuss in detail some of the related work which involves supporting multiple consistency models and show how our mixed consistency model differs from other work. After that, we introduce some of the distributed data sharing systems that have been previously developed. Finally, we discuss techniques in information dissemination literature which can be used to disseminate updates for maintaining consistency.

2.1 Consistency Models

Two orthogonal dimensions of consistency, *timeliness* and *ordering*, are explored in [8].

Ordering determines the value that should be applied/returned by the consistency protocols. When operation ordering is concerned, a number of consistency models have been presented in the distributed shared memory literature.

- Sequential consistency (SC) [52], [13], [68] requires all operations appear to be executed in a serial order that respects the "read-from" ordering and "process" ordering.
- Linearizability [42], [69] is a stronger model than sequential consistency in that it also requires the operations to respect the real timestamp ordering besides the sequential consistency requirements.
- Lamport considered single writer shared objects (called *registers*) in [53] and [54]. He proposed three classes of registers, namely *safe*, *regular* and *atomic*. Each class essentially defines an ordering requirement:

  - Safe registers: a read operation must return the most-recently written value of that register, if it does not overlap a write. Otherwise, the returned value can be arbitrary value allowed by the register.
  - Regular registers: must satisfy safe requirement plus if a read overlaps a sequence of writes, it returns either one of the values written by those writes, or the value before all those writes.
  - Atomic registers: must satisfy regular requirement plus if a read X precedes another read Y, Y cannot return an "older" value than X.

- Causal consistency (CC) [7] is a weaker consistency model because it does not require all replicas to agree on a unique execution history. In particular, causal consistency does not guarantee the ordering of concurrent writes. Raynal and Schiper [68] show that SC is strictly stronger than CC by showing that CC + *"total order on all writes on all objects"* = SC.

- Other orderings such as processor consistency [5], [39] and PRAM (Pipelined Random Access Memory) consistency [55] have been proposed in the distributed shared memory literature. They are easy to implement and provide programmers with relaxed ordering guarantees.

Other ordering guarantees (e.g. lazy release consistency [47] and epsilon serializability [30]) have also been explored in the database and distributed systems literature. They explore different aspects (e.g. synchronization, transaction) of distributed computations and how additional facilities can be used to enhance consistency models. Works on quasi copies [15], [9], lazy replication [51] and bounded ignorance [49] have been focused on controlling inconsistency and relaxing serializability requirement in a replicated database environment. One-copy-serializability is the most common consistency guarantee provided by replicated database algorithms, such as the adaptive data replication algorithm in [77]. The work in [58] focuses on keeping database consistency when schema evolves. They are not closely related to our work because we do not address transaction, atomicity or recovery in mixed consistency.

Timeliness decides how fast a newly created value for an object should become available at various clients. Maintaining web content consistency [57] and δ-time consistency [74] suggest that the value returned can be stale by no more that δ time units. In [50], an efficient implementation of timed consistency based on combined "push" and "pull" techniques is presented.

Our work fits into the operation ordering dimension of consistency model. The mixed consistency model tries to simultaneously allow different order guarantees to meet application needs in wide area heterogeneous environment. We choose SC and CC as the example of strong and weak consistency to define the mixed consistency model in this thesis.

Designing protocols to implement consistency models can benefit from research work on failure detectors [20]. In particular, Ω failure detector [4] has been proposed to allow non-faulty processes to eventually agree on a common leader in an asynchronous environment where processes can crash. We will discuss in Chapter 4 that how mixed consistency protocol handle failures and why Ω failure detector is not suitable to be used in our protocol.

We now discuss some of the work in the literature that is particularly close to ours. It generally involves support for multiple consistency models in the system.

2.1.1 Mixed Consistency

Agrawal et. al. first introduced the term *mixed consistency* in [3] to refer to a parallel programming model for distributed shared memory systems, which combines causal consistency and PRAM [55] consistency. Four kinds of explicit synchronization operations: read locks, write locks, barriers and await operations are provided in their model. In this thesis, we use the same term to refer to a new consistency model combining sequential consistency and causal consistency. The access constraints we propose represent a more general approach: PRAM consistency could be integrated into our mixed consistency model with little effort (see Section 3.5). Furthermore, we do not rely on synchronization mechanisms such as locks and barriers to characterize the model.

2.1.2 Hybrid Consistency

Attiya and Friedman introduced the concept of *hybrid consistency* in [12]. In this model, all read and write operations on shared objects are categorized as either *strong* or *weak*. All processes agree on a sequential order for all strong operations, and on the program order for any two operations issued by the same process in which at least one of them is strong. It does not guarantee any particular order of any two weak operations between two strong operations. In our model, an operation is weak or strong depending on whether it is executed with a strong or weak replica. We define access constraints to develop the mixed consistency model where we do not assume that strong operations can be used to establish order among weak operations when necessary.

2.1.3 Interconnection of Causal Memories

Fernandez, Jimenez, and Cholvi proposed a simple algorithm to interconnect two or more causal consistency systems into one causal consistency system through *gates* in [35]. They further defined a formal framework to describe the interconnection of distributed shared memory systems in [43] and showed that only fast memory models can be interconnected, where read and write operations return immediately after only local computations (i.e. no global synchronization, which is required to implement SC systems). In our mixed consistency model, we take a different access constraints based approach to combine SC and CC together. We do not try to interconnect SC and CC through *gates*, where all the communications between two systems must flow through. Therefore, our result does not contradict theirs.

2.1.4 Continuous Consistency

Yu and Vahdat presented a *conit-based continuous consistency* model for replicated services in their paper [79]. In this model, applications use three measurements of *numerical error*, *order error* and *staleness* as bounds to quantify their relaxed consistency requirements. Those measurements can form a wide spectrum of relaxed yet bounded consistency requirements. The authors used relaxed consistency model with bounded inconsistency

to balance the tradeoffs between performance, consistency and availability. By exploring application and system resource heterogeneity, we also suggest that replicated distributed services should be able to support more than just strong consistency. However, instead of relaxing strong consistency requirements, we believe that certain applications can benefit from having both strongly and weakly consistent replicas at the same time.

Most of the work above focuses on providing the same consistency guarantee to all replicas and their implementations largely ignore the heterogeneity of application behavior and underlying system resources. Instead of providing relaxed ordering or timeliness guarantees to all the replicas in the system, our work focuses on allowing both strong and weak replicas to co-exist at the same time. We believe that our approach can better meet the needs of users who have different requirements and adapt to different system resource levels.

2.2 Data Sharing in Distributed Systems

Many distributed systems, such as distributed file systems, world wide web (WWW), distributed objects and P2P systems, allow distributed users to access shared data that is either cached or replicated at multiple nodes.

Distributed File Systems: Many operating systems use caching to improve file system performances. Some of the popular distributed file systems are xFS [10], Spring [63], NFS [72], Coda [73], Farsite [2] [17] [29] and Ivy [60].

xFS is a serverless file system, which provides strong ordering by ensuring that a single writer or multiple readers are able to access a file at a given time. It allows client nodes to cooperatively cache file blocks that are accessed. Any node in the system can cache data and supply it to other clients. The location independence of such a configuration with fast local-area networks gives better performance and scalability than the traditional systems.

The *Spring* file system supports cache coherent file data and attributes. It uses the virtual memory system to cache data and keep them consistent. It provides two types of file servers to accomplish the task: one that provide coherent access to files they export, and the other that runs on each machine to provide read and write operations for cached data.

The *NFS* system allows multiple clients to access files located at one server. The server is stateless and does not maintain any information on either the clients or the way the file is cached. Therefore, all modifications of the file must be written back to the server once the file cache is closed at the client side. Under "sequential write sharing" (meaning a file cannot be open simultaneously for both reading and writing at the same time in different clients), each client reads the most recent copy of the file. However, NFS does not provide mechanisms to ensure such "sequential write sharing", which may result in client reading stale data.

Farsite is a scalable, decentralized file system. It uses a set of insecure and loosely coupled machines to implement a P2P file system which is secure and reliable. Updates in Farsite are maintained by lazy propagation and content leases. Strong consistency of the file system is maintained by means of leases. For example, a write/read lease has to be obtained before a client can modify/observe the content of a file. Multiple read leases of the same file is allowed, but only one write lease can be granted with no other leases (whether read or write) granted for the same file.

Ivy is distributed P2P file system which supports both read and writes. It provides NFS-like interfaces to application users. Ivy solely consists of a set of logs, stored in distributed hash table (DHash). When network partition happens, the application specific conflict resolvers are used to resolve the update conflicts.

Coda is a distributed file system that provides server and network failure tolerance. It uses server replication and disconnected operations to achieve those goals. The replication unit in Coda is called a *volume*. A client that accesses the *accessible volume storage group* (AVSG) takes care of the update dissemination and implements the "read one, write all" logic, in order to minimize the burden at server side. Network partition is to some extent tolerated by allowing continued operations on client side cached copy and update resolution is applied when client goes back online and submits its offline updates.

Note that most distributed file systems introduced above need to provide a guarantee that the file is accessed in a way that is the same as if the file is stored in a centralized server. This implies strong consistency requirement. In Ivy and Coda, updates are allowed

even when a system partition happens. These systems depend on conflicts resolution when different partitions are connected back together. Our information sharing system is different from the distributed file system in that it generally can tolerate relaxed consistency requirements based on different user needs. Therefore, to provide a "single copy as if being stored in a centralized server" is not part of our goal. Similarly, to provide support for different consistency levels of the same file in the system at the same time is not the goal of distributed file systems.

World Wide Web: Consistency protocols for web caching are described in [40], [21], [18]. Weak consistency protocols based on time to live (TTL) mechanism are presented in [21] and [40]. However, those weak consistency models focus more on providing better scalability and enhancing system performance rather than preventing clients from reading stale data. A stronger notion of consistency based on invalidation and pull is presented in [18]. Generally the consistency requirements in WWW literature mainly focus on reducing the staleness of the cached copy at the client side. In our system, however, reading stale data is allowed, as long as the updates follow certain order that is guaranteed by the system. We focus more on the ordering aspect of the consistency requirement other than timeliness.

Distributed Objects: *CORBA* [65], [24], [59], [62] stands for Common Object Request Broker Architecture, which is Object Management Group (OMG [65])'s open client-server middleware architecture used for applications to access services across networks. CORBA's replication service [11], [1] supports object replication in order to improve performance and provide a mechanism for fault tolerance. The replicas of the same object are kept in strongly consistent state as the requests are totally and causally ordered across all replicas [1].

Object caching has also been studied in systems such as Thor [56], Rover [44], Bayou [76] and others. The *Thor* system assumes a transaction notion for the operations on cached objects and validates them before they are committed. The *Rover* system was developed to address the problem of bandwidth constraints and possible disconnection of clients. The client's copy will be serialized by the system when the client goes back online (or has enough

bandwidth). The users have to manually resolve any possible conflicts. *Bayou* is a replicated, weakly consistent storage system designed for a mobile computing environment where disconnection is possible. Bayou has focused on supporting application-specific mechanisms to detect and resolve the update conflicts. It proposed two techniques for such support: *dependency checks* and *merge procedures*. In addition, the updates are propagated in Bayou by pair-wise anti-entropy in order to guarantee eventual consistency. To our knowledge, the above distributed objects system all provide the same level of consistency to all replicas in the system. None of them allow for replicas of the same object to be running at different consistency level at the same time.

Peer-to-peer Information Sharing: Peer-to-peer (P2P) data sharing systems have become popular in recent years [46] [61] [38]. Generally data items shared by P2P systems are considered to be fairly static and updates seldom happen. For example, in P2P email system described in [45], email components are stored in distributed nodes, and they support only read and delete operations. No updates by the write operation will be generated (so strong consistency requirement is not needed). If updates do happen, the modified data item is usually considered as a newer version of the old one and both versions co-exist in the system. Typically, the directory service maintains an entry of each version for centralized P2P systems such as Gnutella [38] [23] and new updates do not get propagated among replicated data items in such systems. Updates do get propagated in some P2P systems designed to be "writable", like FreeNet [22] and OceanStore [70]. However, they view P2P systems as a homogeneous system and do not consider the resource and application behavior heterogeneity. Furthermore, consistency guarantees in those systems are limited. When used as an update dissemination mechanism for in P2P systems, our work on mixed consistency model and agile dissemination is more flexible and can provide different consistency guarantees to different peers. Finally, our system supports multiple writers of the same object, while most of the P2P systems (e.g. FreeNet [22] and PAST [71]) do not.

2.3 Anti-entropy Dissemination

Besides providing a mixed consistency model for the application to meet the requirements of heterogeneous users, we also provide an agile transport layer that supports various dissemination techniques to better utilize system resources to meet different user requirements. Our agile dissemination work presented in Chapter 6 is closely related with previous work done in the anti-entropy dissemination area.

Demers et. al. presented anti-entropy and rumor mongering as examples of epidemic processes [28]. They described several randomized algorithms for replicated database maintenance. The basic idea of the epidemic process is to spread out the message to randomly chosen nodes in different rounds. Nodes that receive the message in previous round become spreaders in the next round. The number of nodes that receive the message grows exponentially with successive number of rounds. Similar techniques are used to achieve reliable multicast in [16], and scalable message dissemination in [27].

Our agile dissemination work uses rumor spreading techniques in the transport layer to disseminate updates among some replicas. We use heterogeneity which is inherent in the system to enhance the two phase based protocol in [27]. Our system can dynamically choose from direct sending, rumor spreading, and invalidation/pull techniques according to different application based access patterns and available resource levels. Datta, et. al. also presented a formal probabilistic model [27] , focusing on evaluating the number and size of messages introduced in the algorithm. It can also be used to analyze the behavior of our "push" and "pull" phase. However, our analysis in this thesis is more focused on timeliness guarantees based on resource consumption.

2.4 Summary

We discussed related work in this chapter. We first introduced different consistency models that have been developed in the past. We also described other work that supports multiple models, and discussed how our work relates to them. Finally, we discussed several results from the anti-entropy dissemination literature, which is closely related with our agile dissemination work that is used to support the mixed consistency model. In the next chapter,

we will introduce our mixed consistency model and its implementation.

CHAPTER III

MIXED CONSISTENCY: MODEL AND PROTOCOL

We discuss the mixed consistency model and a protocol that can be used to implement it in this chapter. We first introduce the system model that our consistency model will be built upon. We then formally define the mixed consistency (MC) model and use different examples to illustrate the MC requirements. After that, we give an implementation of the MC protocol, and we finally show that the MC protocol meets both MC correctness requirements.

3.1 Motivational Applications

The following examples give an intuition for the access constraints based approach we propose to construct a system with both SC and CC replicas coexisting at the same time. The details of the constraints will be discussed in Section 3.3.

(1) *Traffic Information Sharing*: In a traffic information sharing system, cars use their onboard communication device to 1) receive critical traffic information updates from command center; and 2) exchange other (including critical) traffic information with cars nearby. The command center gathers inputs from many sensors distributed along the highway system to determine the overall traffic conditions and generates critical updates periodically. In order to lower the latency of generating critical updates, there are usually more than one command centers (for example, one for each region), each can generate critical updates based on the information it receives.

This traffic information sharing system can be implemented as a distributed shared object system, where critical traffic information is stored in an SC object (all replicas are SC copies) *CriticalInfo* and other traffic information is stored in a CC object (all replicas are CC copies) *CommonInfo*. Both objects are shared among the command centers and all the cars. The command centers are implemented as SC processes, so they can write critical updates to *CriticalInfo*. All cars are implemented as CC processes. They cannot "write"

CriticalInfo but can "read" the value of *CriticalInfo* to receive the update as needed (e.g., when their local copy is old). When a car receives a critical traffic information update, it also "writes" the update to *CommonInfo* to allow other cars to share. Therefore, the command centers will not be overloaded because only a small number of randomly chosen cars need to access *CriticalInfo*. Other cars can receive the updates through *CommonInfo*. Cars can also exchange local traffic information by reading and writing *CommonInfo*. Such information is not of interest to the command centers so they do not need read access to *CommonInfo* object.

(2) *Cooperative Education Support*: In a cooperative education support system, instructors, students and TAs share a common set of document objects. For a particular homework object, say *Homework3*, the instructor and TA both have an SC copy and the students each has a CC copy. The instructor assigns the homework by updating the state of *Homework3*, which will be propagated and shared among all the CC copies at the student's side. Students are encouraged to discuss the homework and share their opinions with each other by updating the CC copies of their *Homework3* object. Those discussions will not be monitored by the instructor or the TA. In the mean time, common questions and hints will be posted by the TA to help the students solve the problems. Updates from the instructor and the TA are maintained in sequential order to prevent any confusion, while the student discussions are kept causally consistent in order to handle a large number of students.

3.2 System Model

We consider a replicated object system composed of distributed processes, each of which having a local memory where copies of objects are stored. We assume that one process can only have at most one replica of a particular object in its local memory. Replicas can be removed from the local memory when they are no longer needed or there is a memory shortage. New object replicas can be created as needed. However, we will not discuss the creation and deletion of replicas in this chapter. We will focus our discussion on a system where the locations of object replicas are already determined. In such a system, a process has access to its local copies of objects. Processes can also communicate with each other

through reliable pair wise inter-process communication channels.

Let P denote the process set, where each $p_i \in P$ represents an individual process (assuming $1 \leq i \leq N$). We define that P consists of two disjoint subsets: P_{SC} and P_{CC}. Processes in P_{SC} (P_{CC}) are called SC (CC) processes.

Let O denote the object replica set, where each $o_{ij} \in O$ stands for a replica of object o_j stored in process p_i's local memory (assuming $1 \leq j \leq M$). Clearly for a particular j ($1 \leq j \leq M$), all the $o_{xj} \in O$ ($1 \leq x \leq N$) are initialized from the same object o_j. Similarly, we define that O also consists of two disjoint subsets: O_{SC} and O_{CC}. Replicas in O_{SC} (O_{CC}) are called SC (CC) replicas.

3.3 The Mixed Consistency Model

A process p_i accesses an object o_j by invoking the methods on an object replica o_{ij} in its local memory. We categorize the methods into "read" (r) and "write" (w) operations. The "read" method does not change the state of the object copy, while the "write" method does. We adopt the notation used in [7] to define the consistency model of our system, i.e., operation $r_i(o_j)v$ stands for p_i reading object o_j and returning its current value (or state) v, and operation $w_i(o_j)v$ stands for p_i writing to object o_j a new value v. We also use $r(o_j)v$ and $w(o_j)v$ to denote the operations when who issues the operation is clear or unimportant.

3.3.1 Access Constraints

Mixed consistency aims to meet both differing application needs and the wide range of resource levels that are common in heterogeneous environments. Thus, when a resource poor node chooses to make a weak replica of an object and updates it, it cannot be expected to have enough resources to update the strong replicas of the object at other nodes in a manner that is required by the SC model. On the other hand, SC requires that updates observed by strong replicas must be ordered in a uniform manner. These conflicting goals must be reconciled in the mixed consistency model such that strong replicas attain the desired type of ordering for operations observed by them, while allowing weak replicas to achieve the sharing and efficiency required by them. We achieve these goals by developing a set of access constraints that allow weak replicas to access the values of updates done at

strong as well as weak replicas. Strong replicas can only observe values of those updates that are done at the nodes that maintain strong replicas. This is reasonable because (1) nodes with strong replicas have the necessary resources to enforce the strong ordering, and (2) updates done to weak replicas at resource poor nodes, which do not want to incur the cost of strongly ordering the updates, are not observed by the strong replicas to prevent violations of ordering requirements. This is also consistent with the requirements of applications where nodes with weak replicas either do not update objects that have other strong replicas or values of their updates are only shared with other nodes that have weak replicas. The constraints that define which updates can be observed by which type of replicas can be captured by the following two rules:

- **Rule 1**: A SC process can read and write a SC replica and can only write a CC replica;
- **Rule 2**: A CC process can read and write a CC replica and can only read a SC replica.

Table 1 defines both rules in the mixed consistency model. Each row in Table 1 gives the legal access rights of the particular process group to different object groups.

Table 1: Access Constraints Table

	O_{SC}	O_{CC}
P_{SC}	RW	W
P_{CC}	R	RW

3.3.2 Well-formed Serialization and History

Well-formed Serialization: Let T be an arbitrary set of operations. We say T is *well-formed* if and only if no operation in T violates the constraints defined in Table 1. We call S a *serialization* of T if S is a linear sequence containing exactly the operations of T such that each read operation from an object returns the value written by the most recent write to that object. From now on, when we say a *serialization*, it means a *serialization* of a *well-formed* operation set by default. In other words, a *serialization* always respects **Rule 1** and **Rule 2**.

For example, let's consider a system setting of two processes (p_1, p_2) and three objects (o_1, o_2 and o_3). Both p_1 and p_2 have a local copy of all three objects. p_1, o_{11}, o_{22} are marked as SC, while p_2, o_{21}, o_{12}, o_{13}, o_{23} are marked as CC. We have three operation sets defined as:

$$T_1 = \{w_1(o_3)1, r_1(o_1)1, r_1(o_2)1, r_1(o_3)1, w_2(o_1)2, w_2(o_2)2, w_2(o_3)2, r_2(o_2)2\}$$

$$T_2 = \{w_1(o_1)1, w_1(o_2)1, w_2(o_3)2, r_2(o_1)1, r_2(o_1)2, r_2(o_3)2\}$$

$$T_3 = \{w_1(o_1)1, r_2(o_1)1, w_1(o_2)2, r_2(o_3)2, w_2(o_3)2, r_1(o_1)1\}$$

We can see that T_1 is not *well-formed* because $r_1(o_2)1, r_1(o_3)1, w_2(o_2)2$ violates Table 1. T_2 is *well-formed* but it does not have a *serialization* because $r_2(o_1)2$ returns a value 2 that has never been written to o_1 (assuming each object has an initial value 0). T_3 is *well-formed* and

$$S = \{w_2(o_3)2, w_1(o_1)1, w_1(o_2)2, r_2(o_1)1, r_2(o_3)2, r_1(o_1)1\}$$

can be one *serialization* of T_3.

Projection: Let T be an arbitrary set of operations. We define a projection of T on an arbitrary process group S, denoted as T_S, to be a subset of T, which only contains the operations performed by processes in S. Similarly, we also define a projection of T on an arbitrary object replica group J, denoted as T^J, to be a subset of T, which only contains the operations performed to replicas in J. It is easy to see that $(T_S)^J = (T^J)_S = T_S^J$.

Causal Order: Let A be a complete set of all operations. We define the *program order* $\rightarrow$ on A to be a complete set of $< op_1, op_2 >$ such that both op_1 and op_2 are performed by the same process p and op_1 precedes op_2 according to p's local clock. In this case, we write $op_1 \rightarrow op_2$.

Without loss of generality, we assume that all writes to the same object are uniquely valued. The *writes-into order* $\mapsto$ on A is defined as such that $op_1 \mapsto op_2$ holds if and only

p_1 (SC):	w(x)1	r(x)1	w(y)1	r(x)2		
p_2 (SC):	w(x)2		r(x)2	w(y)2	r(x)2	
p_3 (CC):	w(y)3	r(y)4		r(x)1	r(x)2	r(y)2
p_4 (CC):	w(y)4	r(y)3	r(x)1		r(x)2	r(y)2

Figure 1: MC Example with $O_{SC} = \{x_1, x_2, x_3, x_4\}$ and $O_{CC} = \{y_1, y_2, y_3, y_4\}$

if there are o_j and v such that $op_1 = w(o_j)v$ and $op_2 = r(o_j)v$. Please note that here op_1 and op_2 can be performed on different replicas of the same object o_j.

A *causal order* $\Rightarrow$ induced by both $\rightarrow$ and $\mapsto$ in our model is a partial order that is the transitive closure of the program order and the writes-into order defined on A. To be specific: $op_1 \Rightarrow op_2$ holds if and only if one of the following cases holds:

- $op_1 \rightarrow op_2$ (program order); or
- $op_1 \mapsto op_2$ (writes-into order); or
- there is another op_3 such that $op_1 \Rightarrow op_3 \Rightarrow op_2$

History: We define the *global history* (or *history*) of an operation set A, denoted as H, to be a collection of A's operations and the *program* order among those operations, i.e. $H =< A, \rightarrow >$.

3.3.3 Mixed Consistency

We say that a history H is *mixed-consistent* if it satisfies all the following requirements:

1. **SC requirement:** there exists a serialization of A^{SC} such that it respects the program order;
2. **CC requirement:** for each CC process p_i, there exists a serialization of $A_{\{p_i\}} \cup W$ such that it respects the causal order (W denotes the set of all writes).

In order to illustrate the mixed consistency model, let's consider one example. Figure 1 gives the system setting and the execution history H (operations in the same column denote concurrent operations), where the notion x_i (or y_i) denotes the replica of object x (or y) that process p_i has.

The history H in Figure 1 is mixed-consistent because it meets all the requirements:

1. **SC requirement:** we know that

$$A^{SC} = \{w_1(x)1, r_1(x)1, r_1(x)2, w_2(x)2, r_2(x)2,$$
$$r_2(x)2, r_3(x)1, r_3(x)2, r_4(x)1, r_4(x)2\}$$

Clearly A^{SC} is well-formed. And

$$S = \{w_1(x)1, r_1(x)1, r_3(x)1, r_4(x)1, w_2(x)2,$$
$$r_1(x)2, r_2(x)2, r_2(x)2, r_3(x)2, r_4(x)2\}$$

is a serialization that respects the program order of all p_i $(1 \leq i \leq 4)$.

2. **CC requirement:** we have

$$A_{\{p_3\}} = \{w_3(y)3, r_3(y)4, r_3(x)1, r_3(x)2, r_3(y)2\}$$
$$A_{\{p_4\}} = \{w_4(y)4, r_4(y)3, r_4(x)1, r_4(x)2, r_4(y)2\}$$
$$W = \{w_1(x)1, w_2(x)2, w_1(y)1, w_2(y)2, w_3(y)3,$$
$$w_4(y)4\}$$

For process p_3:

$$S_3 = \{w_3(y)3, w_4(y)4, r_3(y)4, w_1(x)1, r_3(x)1,$$
$$w_2(x)2, r_3(x)2, w_1(y)1, w_2(y)2, r_3(y)2\}$$

is the serialization that respects the causal order.

For process p_4:

$$S_4 = \{w_4(y)4, w_3(y)3, r_4(y)3, w_1(x)1, r_4(x)1,$$
$$w_2(x)2, r_4(x)2, w_1(y)1, w_2(y)2, r_4(y)2\}$$

is the serialization that respects the causal order.

3.4 A Mixed Consistency Protocol

To design brand new protocols to ensure either SC or CC is not our purpose. Instead, we show in this chapter that we can combine existing SC and CC protocols together to meet mixed consistency requirements. We choose two well-studied protocols in the literature to achieve this goal.

INITIALIZATION	
NAME	FUNCTIONALITY
init()	Initialize local data structures such as timestamps and queues.
APPLICATION INTERFACE (invoked by upper layer)	
read(x)	Return the value v of object x.
write(x,v)	Write new value v to object x.
LOCAL FUNCTIONS (invoked by local application interfaces)	
write_miss(x,v)	Request a new WRITE_TOKEN from the home node and complete the local write.
read_miss(x)	Request a new READ_TOKEN from the home node and pull new value from the latest writer if necessary.
SYNCHRONOUS COMMUNICATION PRIMITIVES (invoked by remote nodes, block before return)	
value_request(y)	Return the latest value of y (and associated timestamps, if necessary) to the caller node.
write_token_request(y)	Return the new WRITE_TOKEN to the caller node after revoking all other tokens.
read_token_request(y)	Return the new READ_TOKEN to the caller node after revoking other's WRITE_TOKEN, if any.
write_token_revoke(y)	Delete local WRITE_TOKEN.
read_token_revoke(y)	Delete local READ_TOKEN.
ASYNCHRONOUS COMMUNICATION PRIMITIVES (keep running forever, monitoring queues)	
send_daemon()	Keep running forever; send msgs to selected destinations from output queue oqueue.
receive_daemon()	Keep running forever; receive msgs from the input queue iqueue, and apply new values if necessary.

Figure 2: Categorized interfaces of mixed consistency protocol

SC: Home-based Protocol We choose home-based protocol [19] to ensure SC requirements. A designated home node is associated with every object that has a SC replica to coordinate access to such replicas. A node with SC replica must acquire a token prior to access. The home node keeps track of what nodes can read or write the object's replicas and grants READ/WRITE_TOKEN to the nodes that want to read/write this object. The READ_TOKEN is shared by multiple readers, while the WRITE_TOKEN does not coexist with the READ_TOKEN and can only be issued to one writer at a time. Tokens are issued based on a First-Come-First-Serve order observed by the home node. Currently, the location of the home node is randomly assigned in our protocol, although results in [25, 78, 83] suggest that random assignment may have a negative impact on the performance of home-based protocols. We can employ a "smart" home node assignment when an application profile is available.

CC: Causal Memory Protocol We choose vector clock based causal memory protocol [7] to ensure CC requirements are met. When a write operation is performed, a new value along with the local clock will be disseminated to other replicas. The receiver applies the new value when all the "causally preceding" values have arrived and been applied, which is

determined based on the receiver's local vector clock and the clock value that comes with the new object value. In this protocol, the dissemination process can be done in background. Therefore, write operations do not block and return immediately.

3.4.1 Challenges for Mixed Consistency

We give our mixed consistency protocol in Figure 3. The protocol interfaces are summarized in Figure 2, which will be explained later in the chapter to help understand the protocol. Before that, we want to address several problems of integrating the SC and CC protocols together in order to implement the mixed consistency model.

Possible Causal Order Violation: In our model, a process can have both SC and CC replicas. If we simply run the two protocols to maintain replicas based on their type, we could have a potential CC violation. For instance, suppose there are two processes, namely p_1 (SC) and p_2 (CC). Each has a SC copy of o_1 and a CC copy of o_2. p_1 writes o_2 first and then writes o_1. p_2 reads o_1 and then o_2. The home-based protocol can guarantee that p_2 reads what p_1 writes to o_1. But when p_2 reads o_2, it might get the "old" value because p_1's new value for o_2 is possibly still on the way because of the background dissemination of the causal memory protocol. We overcome this obstacle by delaying CC reads (setting the replica as NOT READABLE) if a potential CC violation is possible. In Figure 3, function $read\_miss()$ evaluates the vector clock returned by process p_k and sets any CC object as NOT READABLE if a causal order violation is possible as described above. New CC values will eventually arrive so the delayed CC reads will eventually return with the correct value. In Figure 3, function $receive\_daemon()$ resets the object as READABLE when it receives a new CC value.

Possible Vector Clock Error: When a SC process writes a SC object, the new value will be disseminated to the CC replicas held by other processes. The receiver needs a correct vector clock in the message to order the update to the CC replica (i.e. when to apply the new value). Therefore, all SC replicas should maintain a "correct" vector clock associated with them, which in turn requires that when a CC process writes a CC object, although the value does not propagate to SC replicas (otherwise violates **Rule 1**), the vector clock does.

Our design enforces the dissemination of vector clocks as required. In Figure 3, function *send_daemon*() sends the message to all processes (both SC and CC) when the local writer p_i is a CC process. Please note that this does not violate our access constraints because the function *receive_daemon*() at the receiver's side only applies the value to the local copy when the receiver is a CC process. If the receiver is a SC process, only the vector clock is used to keep the local vector clock correct. The CC value embedded in the message is discarded.

3.4.2 Protocol Interfaces

The protocol interface is shown in Figure 2, where we categorize the functions into initialization, application interface, local functions, synchronous and asynchronous communication primitives. We use the term "synchronous communication" to refer to blocking, RPC style communication. It is performed through Remote Method Invocation (RMI). We use "asynchronous communication" to refer to non-blocking, send/receive daemon style communication. Our protocol does not place any specific requirements on how to perform asynchronous dissemination. Various multicast or rumor-spreading based techniques can be used as the communication layer support for our protocol. We are exploring an adaptive communication layer under our protocol, featuring better performance and bandwidth utilization in a heterogeneous environment.

In our protocol, $read(x)$ and $write(x, v)$ are the interfaces exposed to the applications. These two functions first evaluate if the constraints in Table 1 are violated. If yes, an exception will be thrown. For a SC replica write shown in Figure 4 and 5, $write(x, v)$ triggers a $write\_miss(x, v)$ call if a WRITE_TOKEN is missing, which in turn asks the home node for a write token by calling $write\_token\_request(x)$. Old or conflicting tokens are revoked by $write/read\_token\_revoke(x)$. A SC replica read, shown in Figure 6, can triggers a $read\_miss(x)$ if a READ_TOKEN is missing. When the home node issues the READ_TOKEN, it also tells the reader where the latest copy is (the latest writer) and the reader calls $value\_request(x)$ to fetch the copy. Vector clocks are also returned along with the latest copy by the latest writer in order to ensure the cross SC/CC causal relationship

```
Assume there are N processes in the system,
  i.e.  p1,..., pN.
O: array of M objects.  x:  object name.
  i.e.  O[x] contains value v of object x.

  init()
    //initializing the meta data
    for every CC object x do
        set x as READABLE;
    for every object x do
        for j = 1 to N do
            tx[j] = 0; //initialize the timestamp
    oqueue = <>;
    iqueue = <>;

  write(x, v)
    //write new value v to object x
    if x ∈ SC and pi ∈ SC
        if pi has x's WRITE_TOKEN
            tx[i] = tx[i] + 1;
            O[x] = v;
            enqueue(oqueue, <i,x,v,t>);
        else write_miss(x, v);
    else if x ∈ CC
        tx[i] = tx[i] + 1;
        O[x] = v;
        enqueue(oqueue, <i,x,v,t>);
    else throw WRITE_EXCEPTION;

  read(x)
    //read object x's value
    if x ∈ CC and pi ∈ CC
        if x is NOT READABLE
            wait until x is READABLE;
        return O[x];
    else if x ∈ SC
        if pi has x's READ_TOKEN
            return O[x];
        else return read_miss(x);
    else throw READ_EXCEPTION;

  write_miss(x, v)
    //wait for a WRITE_TOKEN from x's home node pj
    calls pj.write_token_request(x);
    set ''pi has x's WRITE_TOKEN'';
    tx[i] = tx[i] + 1;
    O[x] = v;
    enqueue(oqueue, <i,x,v,t>);

  read_miss(x)
    //wait for a READ_TOKEN from x's home node pj
    call pk=pj.read_token_request(x);
    //assume pk has the latest value of x
    set ''pi has x's READ_TOKEN'';
    if pk is null
        return O[x];
    else
        call <O[x], s>=pk.value_request(x);
        for any CC object z do
            if (∃j ≠ i:sz[j]>tz[j])
                set z as NOT READABLE;
                set condition(z)=sz[j];
        if (sx[i]>tx[i])
            tx[i]=sx[i];
        return O[x];

  value_request(y)
    //assume it is called by process pj
    //return the entire timestamp array
    return <O[y], t>;

  write_token_request(y)
    //assume it is called by process pj
    if y has a WRITE_TOKEN issued to pm
        call pm.write_token_revoke(y);
        return; //grant pj with y's WRITE_TOKEN
    if y have any READ_TOKENs issued
        for any process pn being issued
            call pn.read_token_revoke(y);
        return; //grant pj with y's WRITE_TOKEN
    return;

  read_token_request(y)
    //assume it is called by process pj
    if y have any READ_TOKENs issued
        //assume pk is the last process that
        //had y's WRITE_TOKEN
        return pk; //grant pj with y's READ_TOKEN
    if y has a WRITE_TOKEN issued to pm
        call pm.write_token_revoke(y);
        return pm; //grant pj with y's READ_TOKEN
    return null;

  write_token_revoke(y)
    //assume it is called by process pj
    wait for outstanding writes to y are finished;
    delete y's WRITE_TOKEN;
    return;

  read_token_revoke(y)
    //assume it is called by process pj
    wait for outstanding reads to y are finished;
    delete y's READ_TOKEN;
    return;


  send_daemon()
    //sending daemon, keeps running forever
    if oqueue ≠ <>
        //let MSG = dequeue(oqueue)
        if pi ∈ CC
            disseminate MSG to all other processes;
        else // pi ∈ SC
            disseminate MSG to all CC processes;

  receive_daemon()
    //receiving daemon, keeps running forever
    if iqueue ≠ <>
    //let <j,x,v,s> = head(iqueue) be the msg from pj
        if ((∀h ≠ x and ∀k ≠ j:sh[k]≤th[k])
          AND (sx[j]=tx[j]+1))
            dequeue(iqueue);
            tx[j] = sx[j];
            if pi ∈ CC
                O[x] = v;
            if ((x is NOT READABLE)
              AND (condition(x)==sx[j]))
                set x as READABLE;
```

Figure 3: The Mixed Consistency Protocol - at process p_i

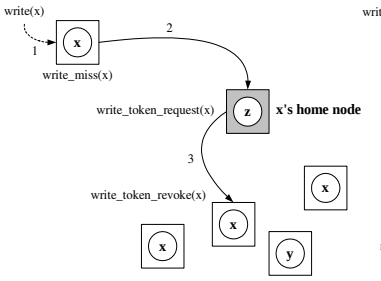

Figure 4: A sample illustration of write(x, v) (x is SC) with write token revocation.

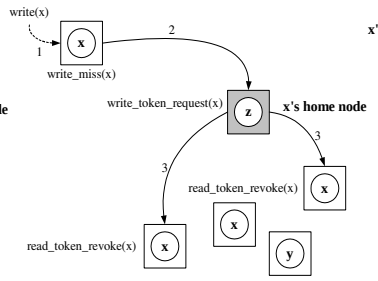

Figure 5: A sample illustration of write(x, v) (x is SC) with read token revocation.

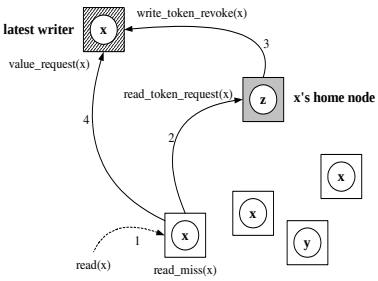

Figure 6: A sample illustration of read(x) (x is SC) with write token revocation.

is correct.

For a CC replica write, a message is constructed and inserted into the outgoing queue (*oqueue*). Function *send_daemon*() will eventually send out the message and the destination side *receive_daemon*() will eventually receive it. A CC replica read is returned immediately if the replica is READABLE. As restricted by the constraints of Table 1, writes to a CC replica will not be propagated to SC replicas. However, we do propagate the meta data (vector clock, to be specific) from CC replicas to SC replicas when a write to CC replica happens. This ensures that the vector clocks on SC replicas can correctly capture the causal relationship even though they do not share the CC values.

3.4.3 Correctness of the Protocol

The protocol given in Figure 3 is correct because we can show that both SC and CC requirements are met in this protocol.

(1) *If there are only SC (CC) replicas in the system*, the protocol behaves the same as the home-based protocol [19] (causal memory protocol [7]) does. Both SC and CC requirements are met.

(2) *If there are mixed SC and CC replicas in the system*, the SC requirements are not violated because the definition of access constraints isolates the writes to CC replicas from SC replicas. These writes, which are not sequentially ordered, are not visible to SC replicas. Now we are trying to argue that CC requirements are not violated either. Let's suppose CC requirements are violated. There are two possibilities: either the causal order

among CC objects is broken, or the causal order among SC and CC objects is broken. The correctness of the causal memory protocol guarantees that the first case does not happen. Our solutions proposed in Section 3.4.1 prevent the second case from happening. So we can be sure that causal order is correctly maintained in our protocol. Therefore, both SC and CC requirements are met.

3.5 Mixed Consistency with Other Models

We use sequential consistency and causal consistency as the strong and weak consistency models to illustrate the access constraints and mixed consistency model. The reason we chose these two is that both models were well studied in the literature (therefore there are many implementation protocols to choose from), and that sequential is proven to be strictly stronger than causal [68]. However, our access constraint based approach and the mixed consistency model is not limited to just combining Sequential and Causal Consistency models. Other consistency models can be combined using similar techniques.

PRAM consistency [55] requires that all the processes in the system see the writes from one process in the order they were issued by that process. But writes from different processes can be seen in a different order by different processes. Therefore, PRAM consistency is weaker than causal consistency. We can integrate PRAM consistency into our mixed consistency model. The modified access constraint table is given in Table 2.

Table 2: Access Constraints Table (SC, CC and PRAM)

	O_{SC}	O_{CC}	O_{PRAM}
P_{SC}	RW	W	W
P_{CC}	R	RW	W
P_{PRAM}	R	R	RW

The protocol that implements PRAM consistency relies on a per-process write operation counter. The process increments the counter when executing local write operations. The new value together with the counter will be disseminated to the other processes in a way that is similar as the causal memory protocol. Upon receiving an update message, the receiver will check the counter in the message and apply the value if it is the "next" write from the sender. Since we use a per-object vector clock at each process in our mixed

consistency protocol, it is trivial to deduce the correct write counter value from the clock for that process. Therefore, the implementation of mixed consistency with PRAM support does not require major changes to the existing protocol.

Linearizability [42], [69] is stronger than sequential consistency in that it also requires all operations to respect the real time ordering in addition to the SC requirements. It can be integrated into our mixed consistency model as well. The new access constraints table is shown in Table 3.

Table 3: Access Constraints Table (Linearizability, SC and CC)

	O_{LN}	O_{SC}	O_{CC}
P_{LN}	RW	W	W
P_{SC}	R	RW	W
P_{CC}	R	R	RW

There are different protocols that implement linearizability under different system assumptions. One straightforward approach is to dedicate a central server that issues read/write token on a global base. Before accessing any object replica in the local memory, every process needs to request the token from the server. The server ensures that only one process can write its local memory at any given time. The update is made available to other processes before the writer releases its write token. In our current MC protocol, we use a dedicated per-object homenode to implement SC protocol, which means object A and B may have different homenodes in the system. In order to combine the linearizability protocol, we need to implement a global homenode for all the objects in the system and update our rules to assign/revoke tokens. We have started working on this model, and the detail will be part of the future work. Another interesting future problem is to explore the possibility of combining Linearizability, SC, CC and PRAM together.

It is also possible to combine Lamport's safe, regular and atomic registers model [53], [54] together using our access constraints based approach, based on the fact that atomic is stronger than regular which is stronger than safe. The access constraints table is shown in Table 4. To explore the implementation of such a mixed consistency model is part of the future work.

Table 4: Access Constraints Table (Atomic, Regular and Safe models)

	O_{Atomic}	$O_{Regular}$	O_{Safe}
P_{Atomic}	RW	W	W
$P_{Regular}$	R	RW	W
P_{Safe}	R	R	RW

3.6 Summary

In this chapter, we introduced the mixed consistency model, which combines both SC and CC requirements together at the same time. Instead of designing new protocols, we gave an implementation based on well-known SC and CC implementations with little modification. We showed that our mixed consistency system can support both SC and CC replicas of the same object at the same time. This gives great flexibility to the application that uses the mixed consistency model. In order to introduce the basic idea of mixed consistency, we assume a failure-free environment in this chapter. We will relax this assumption in Chapter 4 and discuss how to construct mixed consistency protocols that can tolerate failures.

CHAPTER IV

NON-RESPONSIVENESS TOLERANT CAUSAL CONSISTENCY

Starting from this chapter, we consider the effect of failures on consistency protocols. We first show that the causal consistency protocol in our mixed consistency model can tolerate crash failures in an asynchronous environment. In the next chapter, we will address how the mixed consistency model can provide new ways to tolerate failures in distributed systems.

In this chapter, we begin with introducing our new system model, which has crash failure and asynchronous communication assumptions. We then discuss why traditional failure detector (Ω failure detector [4] to be specific) is not sufficient for this model. After that, we introduce a new concept of responsiveness and illustrate how to construct a "non-responsiveness" detector in our system. Based on this new concept, we conclude this chapter by introducing two Non-Responsiveness Tolerant Causal Consistency (NRTCC) protocols. The first one makes use of matrix clock and results in higher overheads. The second one is based on vector clock and has better communication scalability.

4.1 System Model

From now on, we are considering a system consists of n processes, i.e. $P = \{p_1, p_2, ..., p_n\}$, and m distinct objects, i.e. $O = \{o_1, o_2, ..., o_m\}$. For simplicity, we assume a fully replicated scenario, where each process maintains a local copy of each object.

- Processes communicate with each other through pair-wise **FIFO** channels.
- All communication channels are **asynchronously reliable**: There is an unknown maximum message delay δ such that if process p sends a message m to process q at time t and q is a correct process, q receives m from p by time $t + \delta$. We will explain why we need the *reliable* assumption later in Section 4.3.1.
- Possible **crash** failures of processes: Processes can crash arbitrarily during execution and once they crash, they stop responding to/sending any messages and stay as

crashed forever (permanent crash).

4.2 Ω Failure Detector

```
initial_Ω()
  //Initialization for Ω failure detector
  for every process p_j where j ≠ i
      set TIMEOUT_j;
      start timer(j);

send_daemon_Ω()
  //receiving daemon for Ω, keeps running forever
  while (TRUE) do
      send message <ALIVE, p_i> to every process every t time;

receive_daemon_Ω()
  //receiving daemon for Ω, keeps running forever
  upon receive <ALIVE, p_j> do
      ALIVE_i = ALIVE_i ∪ {p_j};
      reset TIMEOUT_j;

timer(int j)
  //Gets called after every unit of time
  TIMEOUT_j--;

timeout_handler(int j)
  //Gets called when timer of process j runs out, i.e., when TIMEOUT_j=0
  upon time out p_j do
      ALIVE_i = ALIVE_i − {p_j};
      TIMEOUT_j++;
      restart timer(j);
```

Figure 7: The Ω failure detector - at process p_i

Ω failure detector has been proposed to detect crash failures in asynchronous distributed systems. The algorithm is described in Figure 7. It guarantees that if a process q crashes, **eventually** $q \notin ALIVE_p$ for every correct process p in the system. However, it does not guarantee at any moment that all processes in the ALIVE set are correct, nor does it guarantee that any processes not in the ALIVE set are faulty. So when a process p equipped with Ω failure detector finds that q has fallen out of its $ALIVE$ set, it can not assume q has failed. Therefore, even though Ω failure detector guarantees the eventual detection of crashed processes, it is hard to use in actual protocols that must decide when a process fails in order to take certain actions.

4.3 Responsiveness

A process can be slow but still correct in a distributed system. Actually, the fundamental difficulty of dealing with crashes in an asynchronous system is because we cannot distinguish

a crashed process from a slow one. If we keep slow processes in the system, it often results in poor system performance. We can hope slow processes will soon become normal again but a process being slow is sometimes not just temporary. For example, it may be caused by a physical memory resource limitation (e.g., gets drained out over time by a memory leak). Therefore, a slow process has a much higher than normal chance of becoming a crashed process in this case. It is thus reasonable to have a mechanism to detect not only crashed but also "slow" processes in the system. Also, often the performance of the system can be improved when non-responsive processes are excluded from the execution of the protocol. In order to better deal with crashed or slow processes, we introduce the concept of **responsiveness** (RES). Formally, we define RES as a binary relationship on process group P, i.e. $RES \subseteq P \times P$. It has the following properties:

- RES is self-reflective, meaning $\forall p \in P$, $(p, p) \in RES$.
- RES is symmetric, meaning $\forall p, q \in P$, if $(p, q) \in RES$ then $(q, p) \in RES$.
- RES is not transitive, meaning $\forall p, r, q \in P$, if $(p, r), (r, q) \in RES$, it does not necessarily mean $(p, q) \in RES$.

We understand that the symmetry assumption may not always be true in real systems. For example, it is not uncommon in today's Internet that packets sent from process p to q are routed via a different path from those sent from q to p. Therefore, it is quite possible that one process may still be able to receive packets from the other but not vice versa. However, we think it is meaningful to assume that RES is symmetric in our case, because 1) it still captures a lot of the node failure cases where a process is unable to receive or send messages; 2) we are focusing on the result not the cause of the failure in practice. Our MC protocol implementation requires a RPC style communication among processes (i.e. a process blocks after sending out a request until a reply has been received). As a result, no matter whether the request or the reply message gets lost in communication, the effect is the same, i.e. the communication between the sender process and the receiver has broken and any operations that require the cooperation of these two processes will fail. Therefore, by assuming that RES is symmetric, we can focus on how to make our protocol respond

to such failures in the system rather than to determine the actual cause of the failure. A system level diagnosis may help determine the root cause of the failure, but that is beyond the scope of this thesis.

We define a process p's **responsive set** RS_p to be a set of processes, such that $\forall q \in RS_p$, one of the following conditions must be met:

- $(p, q) \in RES$.
- $(p, q) \notin RES$, but $\exists r_1, r_2, ...r_k \in P$ $(1 \leq k \leq n)$, such that $(p, r_1) \in RES$ and $(r_1, r_2) \in RES$ and, ..., and $(r_k, q) \in RES$.

We say a process p is **responsive**, if $\forall q \in P$ such that $q \neq p$, we have $(p, q) \in RES$. We say a process p is **non-responsive**, if $\exists q \in P$ such that $q \neq p$, we have $(p, q) \notin RES$ (by the symmetric property, we know for sure that $(p, q) \notin RES$ either). We name the set of all responsive processes R, and the set of non-responsive processes NR. Clearly $R \subseteq P$, $NR \subseteq P$ and $R \cup NR = P$. We know that a responsive process is guaranteed to be correct. A crash failure can turn a responsive process into a non-responsive one, while simply being slow can also do the same. Formally, if C stands for the set of crashed processes, we should have $C \subseteq NR$. Therefore, if we can detect and exclude all non-responsive processes in our protocols, we can guarantee to exclude all crashed processes. The price to pay is that we can possibly also exclude correct but slow processes.

In practice, we use a Non-Responsive Process Detector (NRPD) to detect non-responsive processes in our system.

4.3.1 Non-Responsive Process Detector

There are two parameters in the NRPD algorithm (given in Figure 8) we describe here, i.e. the initial timeout value T and responsive threshold N. Each process is equipped with the detector and it periodically sends out ALIVE messages to all other processes. If timeout happens before such a message is received, the sender will be put into SUSPECTED set, meaning it could potentially be non-responsive. If it failed to meet the timeout for N consecutive times, the sender then will be regarded as non-responsive and will be put into

```
initial_NRPD()
  RESPONSIVE = P; (all processes in the system)
  SUSPECTED = {};
  NONRESPONSIVE = {};
  for every process p_j where j ≠ i
      TIMEOUT_j = OLD_TIMEOUT_j = T;
      THRESHOLD_j = T;
      start timer(j);

send_daemon_NRPD()
  //sending daemon for NRPD, keeps running forever
  while (TRUE) do
      send message <ALIVE, p_i> to every process every T time, unless stopped by other functions;

receive_daemon_NRPD()
  //receiving daemon for NRPD, keeps running forever
  upon receive <ALIVE, p_j> do
      NONRESPONSIVE = NONRESPONSIVE − {p_j};
      resume sending ALIVE messages to p_j;
      SUSPECTED = SUSPECTED − {p_j};
      reset THRESHOLD_j = N;
      reset TIMEOUT_j = OLD_TIMEOUT_j;
      restart timer(j);

timer(int j)
  //Gets called after every unit of time
  TIMEOUT_j--;

timeout_handler(int j)
  //Gets called when timer of process j runs out, i.e., when TIMEOUT_j = 0
  upon timeout of p_j do
      THRESHOLD_j--;
      RESPONSIVE = RESPONSIVE − {p_j};
      SUSPECTED = SUSPECTED + {p_j};
      if (THRESHOLD_j == 0) do
      //Process j is non-responsive
          NONRESPONSIVE = NONRESPONSIVE + {p_j};
          stop sending ALIVE messages to p_j;
          return;
      TIMEOUT_j = 2 * OLD_TIMEOUT_j;
      OLD_TIMEOUT_j = TIMEOUT_j;
      restart timer(j);
```

Figure 8: The Non-Responsive Process Detector (NRPD) Algorithm - at process p_i

NONRESPONSIVE set. In order to tolerate temporary delay of messages, each time a process fails to meet the timeout value, the new timeout value is doubled. The NRPD algorithm is extended from Ω failure detector, which can be viewed as a special case of NRPD with N = 1 (and with both SUSPECTED and NONRESPONSIVE set combined into a potentially failed process group).

The advantage of having both SUSPECTED and NONRESPONSIVE set instead of one NONRESPONSIVE set is to give the application of NRPD an extra degree of flexibility, i.e. the application can choose to use one of the two strategies when trying to deal with non-responsiveness:

- Proactive: The application starts non-responsive handling logic when a process $p_i \in$SUSPECTED $\cup$ NONRESPONSIVE.
- Optimistic: The application starts non-responsive handling logic when a process $p_i \in$NONRESPONSIVE.

The proactive strategy can be used for applications with strict timeliness requirement where responsiveness becomes a critical factor of system behavior. The optimistic strategy can be used for applications with strong liveness requirement where the system should stay as usual for as long as it can. In this thesis, we use the proactive strategy to illustrate our consistency protocols. They can be easily modified to adapt the optimistic strategy.

Please note that the NRPD algorithm can execute correctly without the "reliable" assumption defined in Section 4.1 (i.e. lost heartbeat messages are expected and can trigger one end of the communication link falls out of the RESPONSIVE set of the other end). However, we still need this assumption. The reason is that the MC protocol being built on top of NRPD cannot handle lost *update* messages. If a system can deliver heartbeat messages in time, but can also lose update messages from the application (in real systems where failure detector and application are usually implemented in different processes, such behavior is possible to happen.), the liveness will be broken. For example, let's consider a simple system consists of CC process X, Y and Z. Assume they are exchanging heartbeat messages which all arrive in time. So X, Y and Z is in each other's RESPONSIVE set.

Suppose X executes a write operation and the update message sent to Y is lost. Z receives and applies X's update, and executes another write. Z's new value is received by Y, and now Y needs X's update in order to apply Z's update. However, since X's update message is lost, but X is still in Z's RESPONSIVE set, Z will be waiting for X's update indefinitely. To make NRPD aware of the lost update message will not solve the problem. That is because even if NRPD can put X out of Z's RESPONSIVE set due to the missing update message, X can be put back into the set when the next successful heartbeat message from X arrives.

4.4 Causal Consistency Model

With the presence of non-responsive processes, a causally consistent system is defined as follows: For each process p, there exists a serialization of $A_{\{p\}} \cup W_{RS_p}$ such that it respects the causal order, where $A_{\{p\}}$ is the set containing all operations of process p. W_{RS_p} is the set containing all write operations of processes in RS_p.

4.4.1 A Non-Responsiveness Tolerant Causal Consistency (NRTCC) Protocol

We use the following notions and assumptions when discussing the NRTCC algorithm given by Figure 9:

- Each process p_k maintains a $m \times n$ matrix clock M_k, where $M_k[i, j]$ records the number of writes process p_j performs to object o_i that is known by process p_k.
- We use $M[i, *]$ and $M[*, j]$ to denote the vector clock of row i and column j in the matrix M. They follow the vector clock comparison rules, which are defined in [7].
- Matrix clock comparison rules:
  - $M_1 = M_2$ iff $\forall i \in [1, m], j \in [1, n]$: $M_1[i, j] = M_2[i, j]$.
  - $M_1 < M_2$ iff $\forall i \in [1, m], j \in [1, n]$: $M_1[i, j] \leq M_2[i, j]$ and $\exists a \in [1, m], b \in [1, n]$:$M_1[a, b] < M_2[a, b]$.
  - $M_1 <> M_2$ iff neither $M_1 = M_2$ nor $M_1 < M_2$ or $M_2 < M_1$. We call them concurrent timestamps.

- For simplicity, when there is no confusion, we use M to denote the local clock and S denote the clock in the received messages.

4.4.2 Non-Responsiveness Handling for NRTCC

When a process p_i (its local clock is M) is evaluating a message $m =< j, x, v, S >$ in its *iqueue*, there are three possible cases:

- Case 1: There are no causally proceeding messages of m missing and it contains the "next" update p_i needs (i.e. if $\forall a \in [1, m]$ and $b \in [1, n]$:$S[a, b] \leq M[a, b]$ except $S[x, j] = M[x, j] + 1$): m can be directly applied.

- Case 2: p_i is missing a causally proceeding update of p_j to some objects: It is impossible because of our FIFO reliable channel assumption, which implies that any "previous" updates of p_j must have arrived at p_i, and according to our queue organization rules, those updates must be ordered *before* m and thus must have already been applied.

- Case 3: p_i is missing a causally proceeding update of p_b to some object o_a ($b \neq j$ but a may be x or not), and this update is known by p_j already (i.e. if $\exists a \in [1, m]$ and $b \in [1, n]$ such that $S[a, b] > M[a, b]$): This is possible because p_b could fail in the middle of update dissemination, so that p_j receives the update but p_i does not. However, it could also be caused by the temporary slow communication between p_b and p_i. Therefore, we do not know for sure if p_b has failed or not. If $p_b \in RESPONSIVE_i$, p_i will wait for its update to come (by waiting, we mean p_i can temporarily suspend processing m and continue processing other messages that may be applicable at this moment and it will come back to process m later). If the update finally arrives, the computation will continue. However, if p_i discovers that $p_b \notin RESPONSIVE_i$, there is a chance that the missing update will never come. In that case, p_i will try to contact p_j for the missing update (there is still a chance that the missing update will arrive during the following process, we will discuss it later).

```
init()
  //initializing the meta data
  for each i ∈ [1,m] AND j ∈ [1,n] do
     M[i,j] = 0;
  oqueue = <>;
  iqueue = <>;

write(x, v)
  //write new value v to object o_x
  M[x,i] = M[x,i] + 1;
  o_x = v;
  enqueue(oqueue, <i,x,v,M>);

read(x)
  //read object o_x's value
  if o_x ≠ ⊥
     return o_x;
  else return NOT READABLE;

send_daemon()
  //sending daemon, keeps running forever
  if oqueue ≠ <>
     //let MSG = dequeue(oqueue)
        disseminate MSG to all other processes;

receive_daemon()
  //receiving daemon, keeps running forever
  if iqueue ≠ <>
     //let <j,x,v,S> = head(iqueue) be the msg from p_j
     if ∀a ∈ [1,m] and b ∈ [1,n]:S[a,b] ≤ M[a,b] except S[x,j] = M[x,j] + 1)
        //Got the correct update, apply it
        o_x = v;
        M[x,j] = S[x,j];
        remove <j,x,v,S> from iqueue;
     if ∃a ∈ [1,m] and b ∈ [1,n] such that S[a,b] > M[a,b]
        //Miss an update of p_b to o_a that p_j already knew
        if p_b ∈ RESPONSIVE
           //The missing update will arrive, or p_b ∉ RESPONSIVE eventually.
           do nothing; continue evaluating other messages in iqueue;
        else
           //It is likely that p_b has failed.  Ask for the update from p_j
           request_value(j,b,a,S,x,v);

request_value(j,b,a,S,x,v)
  //Ask from p_j for the update of p_b to o_a w.r.t.  S[a,b]
  send REQUEST message <i,b,a,S> to p_j, wait for reply or until p_j ∉ RESPONSIVE;
  if p_j ∉ RESPONSIVE
     //p_j has possibly failed, the reply may never come.
     remove <j,x,v,S> from iqueue; //discard the update from p_j
  upon receive message REPLY <b,a,o_a,T> from p_j
     insert <b,a,o_a,T> into iqueue;
  upon receive message REPLY <b,a,⊥,T> from p_j
     insert <b,a,⊥,T> into iqueue;

fulfill_value(j,b,a,S)
  //p_j asks for the update of p_b to o_a w.r.t.  S[a,b]
  if ∀c ∈ [1,n]:M[a,c] = S[a,c]
     //The update has not been overwritten yet
     send REPLY message <b,a,o_a,S>;
  if ∃c ∈ [1,n]:M[a,c] > S[a,c]
     //The update has been overwritten by others
     send REPLY message <b,a,⊥,S> to p_j;
```

Figure 9: The NRTCC protocol - at process p_i

Let's assume that the request message is $< i, b, a, S >$. There are three possible scenarios:

1. p_j's local object a has not been overwritten yet, thus its value is still the update of p_b to o_a w.r.t. $S[a, b]$ (i.e. if $M[a, *] = S[a, *]$. Here M is the local clock of p_j upon processing value request message of p_i, S is the clock embedded in the request message, which was p_j's local clock when its update message m got sent to p_i): p_j then sends the reply message $< b, a, o_a, S >$ to p_i, and this message will be put into the p_i's *iqueue* as if it was sent by p_b. Later on, it will be applied by p_i before m because it causally proceeds m. So the computation continues.

2. p_j's local object a has already been overwritten by whichever process(es) (i.e. if $M[a, *] > S[a, *]$), so that the missing update value cannot be retrieved from p_j's local value of a. In this case, p_j will send a reply message $< b, a, \perp, S >$ ($\perp$ is a symbol for *invalid* value) to p_i. p_i will then mark object a as NOT READABLE. However, p_is local matrix clock can be advanced, so the computation may continue. The NOT READABLE value of a will be overwritten by updates to object a in the future. Before that, a read request to a from p_i either temporarily blocks or returns *invalid* value, whichever is desirable to the application.

3. p_j failed and p_i's request will never get replied (eventually p_j will fall out of p_i's $RESPONSIVE$ set): p_i will wait for p_j's reply until $p_j \notin RESPONSIVE_i$. It then drops the original update from p_j, i.e. message $< j, x, v, S >$, because it cannot be applied to local replica since a causally proceeding message is missing.

Note that p_i can fail at any moment during the above processing. Since there is no process waiting for p_i's response, it does not affect the continuity of computation at other processes.

It is possible that the missing update message $g =< b, a, v, T >$ arrives at p_i during the above value request process. This will create a "race" condition like scenario in our algorithm, because p_j's reply message can either beat g at p_i or not. Whichever message that comes "later" will get dropped by p_i. Therefore, if p_j's reply message gets processed by p_i first and it contains $\perp$ value or if p_j falls out of $RESPONSIVE_i$ before g gets processed

(either case will trigger a being marked as NOT READABLE), object a will remain NOT READABLE until the next update overwrites it.

4.4.3 A Scalable Non-Responsiveness Tolerant Causal Consistency (NRTCC-S) Protocol

We use the following notation and assumptions when discussing the NRTCC-S algorithm given by Figure 10:

- Each process p_k maintains a size n vector clock T_k, where $T_k[i]$ records the number of updates (writes) process p_i performs that is known by process p_k.
- For each object replica o_x, process p_k records which process last writes to it ($last\_writer(o_x)$) and what update number that write operation has ($update\_number(o_x)$). For example, if $last\_writer(o_4)$ and $update\_number(o_4)$ return p_2 and 3 respectively, we know that object replica o_4 now has the value that is written by the third update from process p_2.
- Each process maintains a history of updates $\{< p_i, num, o_j >\}$ that have been performed by every process in the system, where p_i's num-th update was performed on object o_j. Therefore, given a process id i and an update number k, a lookup function $history.lookup(i, k)$ can return the id of the object that process modified in that update.
- For simplicity, when there is no confusion, we use L to denote the local clock and S denote the clock in the received messages (i.e. the sender's local clock).

4.4.4 Non-Responsiveness Handling for NRTCC-S

When a process p_i (its local clock is L) is evaluating a message $m =< j, x, v, S >$ in its $iqueue$, there are three possible cases:

- Case 1: There are no causally proceeding messages of m missing (i.e. if $\forall i \in [1, n]$:$S[i] \leq L[i]$ except $S[j] = L[j] + 1$): m can be directly applied.

```
init()
  //initializing the meta data
  for each i ∈ [1,n] do L[i] = 0;
  for each j ∈ [1,m] do
     LAST_WRITER[j] = {}; UPDATE_NO[j] = 0;
  history = {};oqueue = <>; iqueue = <>;

write(x, v)
  //write new value v to object o_x
  o_x = v;
  LAST_UPDATE[x] = {p_i};
  UPDATE_NO[x] ++;
  append(history, <i,UPDATE_NO[x],x>);
  enqueue(oqueue, <i,x,v,L>);

read(x)
  //read object o_x's value
  if (o_x ≠ ⊥) return o_x; else return NOT READABLE;

send_daemon()
  //sending daemon, keeps running forever
  if (oqueue ≠ <>) //let MSG = dequeue(oqueue)
     disseminate MSG to all other processes;

receive_daemon()
  //receiving daemon, keeps running forever
  if (iqueue ≠ <>)
     //let <j,x,v,S> = head(iqueue) be the msg from p_j
     if (∀i ∈ [1,n] : S[i] ≤ L[i] except S[j] = L[j] + 1)
        //Got the correct update, apply it
        o_x = v;
        L[j] = S[j];
        LAST_UPDATE[x] = {p_j};
        UPDATE_NO[x] = S[j];
        remove <j,x,v,S> from iqueue;
     if (∃a ∈ [1,n] such that S[a] > L[a])
        //Miss an update of p_a that p_j already knew
        if (p_a ∈ RESPONSIVE)
           //The missing update will arrive, or p_a ∉ RESPONSIVE eventually.
           do nothing; continue evaluating other messages in iqueue;
        else
           //It is likely that p_a has failed.  Ask for the update from others
           request_value(a,S[a]);

request_value(a,k)
  //Ask for the ''k-th'' update of p_a from everyone
  disseminate VALUE REQUEST message <i,a,k> to every process;

fulfill_value()
  //upon receiving VALUE REQUEST message <i,a,k>
  let x = search(history, a, k);
     if (LAST_UPDATE[x]==a) AND (UPDATE_NO[x]==k)
        send VALUE REPLY message <a,k,x,o_x> to p_i;
  send VALUE REPLY message <a,k,x,⊥> to p_i;

apply_value()
  //upon receiving VALUE REPLY message <a,k,x,v>
  if (k < L[a]) discard the message and return;
  if (k == L[a]) AND (o_x ≠ ⊥)
     discard the message and return;
  o_x = v; L[a] = k;
  LAST_UPDATE[x] = {p_a}; UPDATE_NO[x] = k;
  append(history, <a,k,x>);
```

Figure 10: The NRTCC-S protocol - at process p_i

- Case 2: p_i is missing a causally proceeding update of p_j (e.g. p_i received a 3rd update from p_j, but missed the 2nd one): It is impossible because of our FIFO reliable channel assumption, which implies that any "previous" updates of p_j must have arrived at p_i, and according to our queue organization rules, those updates must be ordered *before* m and thus must have already been applied.

- Case 3: p_i is missing a causally proceeding update of p_a to some object ($b \neq j$), and this update is known by p_j already (i.e. if $\exists a \in [1, n]$ such that $S[a] > L[a]$): This is possible because p_a could fail in the middle of update dissemination, so that p_j receives the update but p_i does not. However, it could also be caused by the temporary slow communication between p_a and p_i. Therefore, we do not know for sure if p_a has failed or not. If $p_a \in RESPONSIVE_i$, p_i will wait for its update to come (by waiting, we mean p_i can temporarily suspend processing m and continue processing other messages that may be applicable at this moment. p_i will come back to process m later). If the update finally arrives, the computation will continue. However, if p_i discovers that $p_a \notin RESPONSIVE_i$, there is a chance that the missing update will never come. In that case, p_i will try to contact other processes for the missing update (There is still a chance that the missing update will arrive during the following process, we will discuss it later).

  Let's assume that the VALUE REQUEST message is $< i, a, S[a] >$. When a process p_k receives such message, there are two possible scenarios:

  1. p_k finds that p_a's $S[a]$-th update has not been overwritten yet (p_k knows this by finding out an object replica x whose last writer is p_a and the update number is $S[a]$). Therefore, p_k retrieves the name and the value (o_x) of that object and sends it back to p_i.

  2. p_k finds that p_a's $S[a]$-th update has already been overwritten (meaning p_k cannot find out an object replica x whose last writer is p_a and the update number is $S[a]$, but p_k knows what object was p_a's $S[a]$-th update made to, say y). Therefore, p_k cannot fulfill the request from p_i. It replies to p_i with an invalid value $\perp$ of y.

After p_i sends out the VALUE REQUEST message, it will receive the missing update (i.e. p_a's $S[a]$-th update) from others, with either the correct value v or an invalidate value $\perp$. p_i will apply the value it receives. If the value is $\perp$, the replica will be placed into UNREADABLE state, until a correct value is applied later, or a new value has overwritten it. It is possible that multiple values are received during the value request process because the REQUEST message was broadcasted and multiple process may reply. In that case, a correct value will always overwrite an invalidate value $\perp$. And duplicate replies with the same value will be discarded.

Noote that p_i can fail at any moment during the above computation. Since there is no process waiting for p_i's response, it does not affect the continuity of computation of other processes.

It is possible that the missing update message $g =< b, a, v, T >$ arrives at p_i during the above value request process. This will create a "race" condition like scenario in our algorithm, because p_j's reply message can either beat g at p_i or not. Whichever message that comes "later" will get dropped by p_i. Therefore, if p_j's reply message gets processed by p_i first and it contains $\perp$ value or if p_j falls out of $RESPONSIVE_i$ before g gets processed (either case will trigger a being marked as NOT READABLE), object a will remain NOT READABLE until the next update overwrites it.

4.5 Correctness of the Protocols

We show in this section that both NRTCC and NRTCC-S implements the causal consistency model with the presence of non-responsive processes defined in section 4.4. There are two problems we need to address in order to argue that the protocols are correct.

1. *Safety: If a process p_j's update m has been applied at p_i, all causally proceeding updates of m observed by p_j must have already been applied at p_i.* Both NRTCC and NRTCC-S use matrix/vector clock to ensure that causally proceeding updates of m from processes that are responsive to p_i should be applied before m. If an update m' is missing at p_i and it is from a non-responsive process p_k(note that although p_k is non-responsive to p_i, it may be responsive to p_j), our two protocols take different

yet similar approaches to solve the problem: in NRTCC protocol, p_i asks p_j for the missing update (there are three different scenarios discussed in Case 3 of NRTCC protocol discussion). If the missing update cannot be recovered, p_i has to drop the update m from p_j because the correctness requirement cannot be violated. In NRTCC-S protocol, p_i asks every responsive processes in system for the missing update (there are also three different scenarios discussed in Case 3 of NRTCC-S protocol discussion). If the missing update cannot be recovered, p_i will drop the update m. Therefore, the correctness of both protocols is ensured.

2. *Liveness: If all causally proceeding updates of m (issued by p_j) have already been applied at p_i, m must be applied at p_i so long as p_j is responsive.* As we can see from both protocols that an update m can only be dropped at p_i if and only if there is a causally proceeding update m' that has not been applied and cannot be recovered. Therefore, we can ensure the liveness of both protocols.

3. *Completeness: All updates that are applied at p_i come either from p_i or from p_i's responsive set.* In both protocols, updates from local process get applied without the need to contact other processes. Updates from other processes will be applied after all causally proceeding updates have been applied, as long as one of the following conditions is met:

   - Updates have been issued by a process that is directly responsive to p_i.
   - Updates have been issued by a process that is in-directly responsive to p_i, meaning there is a chain of "responsive" processes that connects p_i with the update issuer.

   These conditions are the exact definition of responsive set we introduced in section 4.4. Therefore, the completeness of both protocols are also ensured.

4.6 Summary

In this chapter, we considered the problem of how to deal with crash failures in an asynchronous environment for the causal consistency model. We introduced a new concept

responsiveness and constructed a "non-responsiveness" detector in our system. Based on this detector, we constructed two non-responsiveness tolerant causal consistency protocols.

In the next chapter, we will discuss issues related with agile dissemination technique and how the causal consistency protocol can benefit from it in order to better meet users needs under different system resource constraints.

CHAPTER V

NON-RESPONSIVENESS TOLERANT MIXED CONSISTENCY

In this chapter, we discuss how to implement a mixed consistency protocol in the presence of non-responsive processes. We show that our mixed consistency model can support a flexible downgrading and upgrading mechanism to handle non-responsive processes or changes of available system resources or both. For example, when the home node process of an object becomes non-responsive, it is impossible for the SC replicas of that object to continue maintaining SC semantics. The MC protocol enables the scenario where those affected SC replicas can be switched to CC (downgrade). When the home node becomes responsive again, those CC replicas can be switched back to SC (upgrade). Similarly, when a SC process's local resource level is low and it cannot dedicate enough resources to ensure SC consistency, this process is switched to CC so that it consumes less resources for running its consistency protocol. When the resource level is back to normal, it can switch back to SC. The upgrade and downgrade mechanism provides a transparent way of handling non-responsiveness and changes of available system resources. We implement both upgrade and downgrade protocols. Together with the NRTCC protocol discussed in Chapter 4, they make the MC model suitable for building highly available and reliable services in the middleware layer.

5.1 Downgrade SC Replica to CC

We know that the home-based SC protocol requires cooperation from other processes as well as the home node (process) in order to meet the correctness requirement of SC. If any of the processes p in the SC protocol becomes non-responsive, the protocol cannot make progress. However, as discussed in the last chapter, the CC protocol (NRTCC) can still make progress in the presence of non-responsive processes. If we can downgrade the affected SC replica to CC, the whole system can still continue to execute without disruption. Furthermore, since the non-responsiveness is a relationship defined between two processes, when a process p

becomes non-responsive to process q, it may still be responsive to the rest of the system. If the SC protocol at q cannot continue execution because of p, it does not necessarily mean that other processes cannot execute their SC protocol correctly. Therefore, each process in the system can make its own decision about whether to downgrade or not based on its knowledge of what processes are non-responsive to it. To be more specific, here are the cases when replica downgrade should happen while process p_i wants to service a read or write request of SC replica x:

1. *x's home node D_x becomes non-responsive to p_i before proper tokens being obtained.* The original home-based protocol will fail because without obtaining a proper token, neither read nor write operations can proceed.

2. *The process(es) that currently holding a token that needs to be revoked becomes non-responsive to D_x.* This case is similar to case 1 in that p_i cannot obtain the token that is necessary to proceed.

3. *When servicing a write request, the last writer becomes non-responsive to p_i before the new value can be pulled by p_i.* In this case, original home-based protocol will also fail because the new value cannot be pulled from the last writer by p_i when it is servicing the read request. If the last writer becomes non-responsive to the reader, the pull operation cannot succeed. However, p_i can ask other processes for the new value because other SC processes may have already pulled it. If p_i successfully finds the value, the home-based protocol can still function. However, if no such value is found, the protocol will fail.

The downgrade protocol is straightforward, as the *downgrade_replica* function shown in Figure 11.

From the protocol, we know that immediately after downgrading, x carries the value that it pulled from an SC replica the last time p_i serviced a $read(x)$ request. This could be an "old" value because after p_i serviced that read request, there may be many SC write requests to x executed by other processes. Since SC protocol is a pull-based protocol, if p_i does not service another read request before downgrading, the latest value won't be pulled

```
downgrade_replica(x)
  //this function is called by a process that holds an SC copy of x
  //and we want to downgrade it to CC
  mark x from SC to CC;
  send <DOWNGRADE, x, p_i> to everyone;

downgrade_process()
  //this function runs on every SC process
  mark p_i (itself) from SC to CC;
  send <DOWNGRADE, p_i, p_i> to everyone;
```

Figure 11: The Downgrade Protocol - at process p_i

to it. To force p_i to do a value pull before downgrading is not going to work, because by the time p_i decides to downgrade, the SC protocol must have already suffered from the above three cases and thus it cannot successfully pull the "latest" SC value. Fortunately, we show that it is not necessary:

1. *Safety: This currently available value of x at p_i has been seen by other CC replicas of x, therefore it is not a newly introduced value by downgrading, hence does not violate CC requirement.* We know that each update made to a SC replica will be propagated to all other CC replicas, using the same mechanism as what CC protocol uses. Those updates carry correct matrix clock timestamps, so they can be treated just as a CC update at the receiver side. What that means is that to the CC replica "world", SC updates are just CC updates. Therefore, the "initial" value carried by the downgrading replica satisfies the CC requirement.

2. *Liveness: This currently available value of x, even if it is old and carries an old vector timestamp with it, can still be updated (overwritten) by newly issued CC writes on x, hence the CC protocol can function.* The "initial" value of x may be "old", which means it may carry a matrix clock timestamp where the vector indicating x has not been updated since the last time x was pulled. Therefore, when the newly downgraded x takes this value and the timestamp as its starting value and timestamp for CC, we have to make sure that later on new CC updates can be accepted and updated on x according to CC updating rules.

   In general, consider p_i's clock M (right after downgrade), and S from a future CC

update m from process p_j. There are four cases:

- Case 1: There are no causally proceeding updates of m missing and m contains the "next" update p_i is expecting, i.e. $\forall a \in [1, m]$ and $b \in [1, n]$:$S[a, b] \leq M[a, b]$ except $S[x, j] = M[x, j] + 1$: m can be directly applied.
- Case 2: p_i is missing one or more causally proceeding update of p_j to x, i.e. $\forall a \in [1, m]$ and $b \in [1, n]$:$S[a, b] \leq M[a, b]$ except $S[x, j] > M[x, j] + 1$: It is possible because before downgrading, p_j's updates did not get applied at p_i. So these "old" updates to x were lost. But p_i does not have to recover them because x has just been downgraded to CC and as long as m gets applied, it does not violate the *causal relationship*. Therefore, in this case, m can still be directly applied.
- Case 3: p_i is missing one or more causally proceeding updates of p_j to object a ($a \neq x$): This case is possible too, and the reason is the same as in Case 2. There are two possible cases here though:
  - Both p_i and p_j have a SC replica of a. This is possible since the vector clock of SC replica only gets updated when new value is pulled. It is possible that p_j has pulled a new value (thus advanced its clock) but p_i has not. But it is OK to apply m because p_i's clock on a will be updated when servicing a read request on a, and it does not violate the *causal relationship*.
  - p_i has a CC replica of a and it does not matter what replica of a p_j may have. The scenario of a missing update to a is not possible because any updates from p_j to CC replicas must have already been received under our FIFO reliable channel assumption. And according to our queue organization, those updates must have been ordered before m and thus must have already been applied.
- Case 4: p_i is missing a causally proceeding update of p_b to some object o_a ($b \neq j$ but a may or may not be x), and this update is known by p_j already (i.e. if $\exists a \in [1, m]$ and $b \in [1, n]$ such that $S[a, b] > M[a, b]$): This is possible and the

```
upgrade_replica(x, v)
  //this function is called by a process that holds an CC copy of x
  //when it wants to upgrade to SC
  if D(x)∉RESPONSIVE
     throw UPGRADE_HOMENODE_EXCEPTION;
  mark x from CC to SC;
  x = ⊥;
  x = read_miss(x); //read_miss() is defined in MC protocol
  if (x == ⊥)
     //this x is the first SC copy of object x
     if (v≠⊥) x = v;
     else throw UPGRADE_VALUE_EXCEPTION;
  send out message <UPGRADE, x, p_i> to all others;

upgrade_process()
  //this function is called by a CC process
  //when it upgrades to SC
  mark p_i from CC to SC;
  send <UPGRADE, p_i, p_i> to everyone;
```

Figure 12: The Upgrade Protocol - at process p_i

handling can be found in Section 4.4.2 and 4.4.4. We will omit the discussion here.

From the above discussion, we know that the function *receive_daemon*() given in Figure 9 already handles the update in CC protocol. Which means that after downgrading, the liveness of x can be ensured.

5.2 Downgrade SC Process to CC

We also know that SC processes require more resources to ensure strong consistency of replicated data. Therefore, if the process observes that the resources it can utilize for consistency maintenance decreases below a certain level, it can no longer ensure strong consistency. It is then reasonable to downgrade the process from SC to CC in order to lower the consumption of system resources. The downgrade protocol is defined by the *downgrade_process* function shown in Figure 11.

5.3 Upgrade CC Replica to SC

Since the downgrade decision is made by each individual process, it is reasonable that the upgrade decision is also made "locally". However, when a process p decides to upgrade one of its CC replicas, say x from CC to SC, it needs to decide what is a proper starting

value for that SC x. To simply carry over the CC value could violate the SC requirement. There are two options we can choose from. In a general case, when the system already has other SC replicas of x up and running, we simply force this newly upgraded x to fetch the latest value before upgrading completes. The replication system itself cannot decide a proper value when there is no other SC x copies existing in the system. In this case, we rely on the application to supply a starting value. The update protocol is shown in Figure 11 (*upgrade_replica* function).

5.4 Upgrade CC Process to SC

The upgrade of process is similar to downgrading it. We only need to change the process tag, which in turn will affect the access constraints evaluation in the protocol. The upgrade protocol is straightforward, as the *upgrade_process* function shown in Figure 12.

5.5 Summary

Mixed consistency allows downgrade and upgrade of both replicas and processes when desired. Ir provides a new way of handling non-responsive processes and changing levels of available system resources. The protocol for these operations can be easily developed. We give our implementation of both operations in this chapter. Together with the NRTCC protocol discussed in Chapter 4, they make the MC model suitable for building highly available and reliable services at the application layer.

CHAPTER VI

AGILE DISSEMINATION

In previous chapters, we have introduced the mixed consistency model and its implementation. We also discussed how non-responsive nodes are tolerated in an asynchronous environment by our causal consistency protocols. In this chapter, we discuss how to better support our consistency protocols by designing an agile dissemination layer that combines different dissemination techniques and can adapt to different application needs and system resources. We first introduce the system model used to construct our dissemination algorithm. In particular, we explain the replica tagging technique we introduce to capture the heterogeneity of application needs and system resources. We then present the agile dissemination protocols, and briefly show how it can improve the performance in massive information dissemination applications. As a demonstration, we finally present the causal consistency protocol that is built on top of the agile dissemination protocols.

6.1 System Model

In previous chapters, we define our system model based on processes and objects/replicas. Our agile dissemination technique is not restricted to only supporting consistency protocols. In fact, it can be used to support many other distributed services. Therefore, we use the concept of distributed services to introduce our system model for the agile dissemination technique.

We consider services that are implemented using distributed objects which can be replicated at multiple nodes. Requests are serviced at a certain object replica by invoking its methods. We categorize all methods into "read" and "write" methods. The "read" method does not change the state of the object, while the "write" method does. We use the term *object state* to denote the internal state of a shared object. Each individual state change produces a new *object value*. New object values are to be disseminated using the protocol that we design. The term *replica tag* is used in this thesis to refer to the attribute that one

replica has, e.g., we can say that the tag of replica A is *Eager Reader* and *Broad Bandwidth*.

The new object value should be disseminated to other copies in the system. The problem we are focusing on in this thesis is how to adaptively propagate the updates among object replicas with respect to a certain application level consistency requirement in a heterogeneous environment.

We make minimal assumptions about the underlying system and assume an overlay peer-to-peer network system. We do not assume any specific physical topology of the network. All peers in the system are equal and no specialized structure (e.g. ring) exists. However, our algorithm can take advantage of certain topology when such information is available in order to achieve better performance. In this thesis, we present the adaptive object value dissemination protocol on top of this peer-to-peer architecture.

6.1.1 Maintaining Replica Tags

In this section, we discuss how to define the tag of a replica by putting it into replica groups. We call this process "replica tagging", e.g. "a replica R_i is tagged with ER" and "R_i is put into group ER" have the same meaning. In this thesis, we present two orthogonal ways of replica tagging: by read frequency and by network bandwidth.

6.1.1.1 Read Frequency

For any replica R_i, we can collect the number of read requests it services during a certain period of time and the time intervals between any two consecutive read requests. These can be obtained from an access monitor. Suppose in time period t, replica R_i services n_i read requests and the time interval of these requests are $V_i = \{v_{12}, v_{23}, ..., v_{n_i-1,n_i}\}$. We can then tagging R_i by applying the algorithm in figure 13, where $B1_i$ and $B2_i$ are pre-defined bias values. They can be adjusted at run time. In order to better capture the access patterns, different replicas can use different bias values based on their ability to service "read" requests. For simplicity, we assume that the same bias values, $B1$ and $B2$, are used for each replicas.

In this algorithm, n_i is compared against $B1_i$, which serves as a cardinality constraint on the number of read requests serviced; The value $b2_i$ is computed and compared with

```
Tagging_F_r()
  //n_i, V_i are obtained from access monitor
  mean_i = mean value of V_i; //compute mean of V_i
  b2_i = 0;
  for each v_ij in V_i do
      v_ij = v_ij / mean_i ;//Normalization
      b2_i = b2_i + |v_ij − 1|;
  b2_i = b2_i / (n_i − 1);
  if (b2_i > B2_i) then tag R_i as UR; //Unpredictable Reader
  else
      if (n_i > B1_i) then tag R_i as ER; //Eager Reader
      else tag R_i as LR; //Lazy Reader
```

Figure 13: Replica Tagging According to Read Frequency

$B2_i$, which can be considered as a regularity constraint on time interval sequence V_i. We are aware that normally people can use "variance" to capture the regularity of V_i. Our algorithm also works well with variance. However, variance is related to "mean" value. We cannot set one variance bias for different V_is without normalizing the values first, which is impractical to do in real systems. Therefore, we use $b2_i$ instead of variance in our algorithm. We will show that all the $b2_i$s have the same bound of $[0, 2]$. It is especially helpful for us to study the impact of regularity on tagging replica by adjusting one regularity bias value for all replicas.

We here prove the bound of $b2_i$. From the algorithm, we know that

$$b2_i = \sum_{i=1}^{n_i-1} \frac{|\frac{v_{i,i+1}}{\bar{v}} - 1|}{n_i - 1} = \sum_{i=1}^{n_i-1} \frac{|v_{i,i+1} - \bar{v}|}{(n_i - 1) * \bar{v}}$$

It is trivial to know that $b2_i \geq 0$. We show below that $b2_i \leq 2$. Given the fact that all $v_{i,j}$s are positive, we have

$$|v_{i,i+1} - \bar{v}| \leq |v_{i,i+1} + \bar{v}| = v_{i,i+1} + \bar{v}$$

Therefore,

$$\sum_{i=1}^{n_i-1} |v_{i,i+1} - \bar{v}| \leq \sum_{i=1}^{n_i-1} (v_{i,i+1} + \bar{v}) = \sum_{i=1}^{n_i-1} v_{i,i+1} + (n_i - 1) * \bar{v} = 2(n_i - 1) * \bar{v}$$

which concludes $b2_i \leq 2$.

We use traces obtained in a real caching system to demonstrate the effectiveness of the tagging algorithm in figure 13. We show that the heterogeneous reader behavior does exist in real life systems.

The traces were obtained from "ircache" project website (http://www.ircache.net/). The trace consists of files from different physical locations, keeping track of local web cache access records in a certain period of time. The particular result we show in this thesis obtained by running our algorithm on the "sd.sanitized-access.20011212" trace. It contains about 710K records of 137 clients who access 26519 web objects [1]. Because only a small portion of the web objects (721 out of 26519) are accessed by more than 5 clients, we consider them as the objects of interest and discard the rest of the objects. Those "hot" web objects are more likely to be replicated in real life systems. Based on the ircache trace, we compute the percentage of ER, LR and UR replicas given the condition that "hot" objects are replicated among those clients who have accessed them. We plot our results accordingly. For example, if a web object WO_1 is replicated at 50 clients, and the number of ER, LR, and UR replicas is 5, 13 and 27 (10%, 36%, and 54%) respectively. WO_1 is then categorized into "less than 20%" for ER, "between 20% and 40%" for LR and "between 40% and 80%" for UR.

We ran the algorithm under different combinations of B1 and B2 settings. We chose B1 from {5, 20}, and B2 from {0.1, 0.2, 0.3, 0.4}. The results are shown in figure 14 and 15. Figure 14 shows the results computed when B1=5, and figure 15 is computed when B1=20. In each figure, 4 different B2's are chosen to show the result. In all figures, the X-axis shows 5 categories. "0.2" stands for "less than 20%", "0.4" stands for "between 20% and 40%" and so on. The Y-axis shows the number of replicas falls into each category.

From figure 14 and 15, we find out that in most cases (700+ out of 721), the percentage of ER replicas of a particular "hot" web object is under 20%. 500+ out of 721 web objects have more than 80% replicas belong to UR and 100+ web objects have between 60% and 80% replicas belong to UR. For LR replicas, 500+ out of 721 have a percentage below 20%, and 100+ out of 721 are between 20% and 40%. We observe the fact that for a given

[1] we categorize the URLs into one web object if they have the same address prefix.

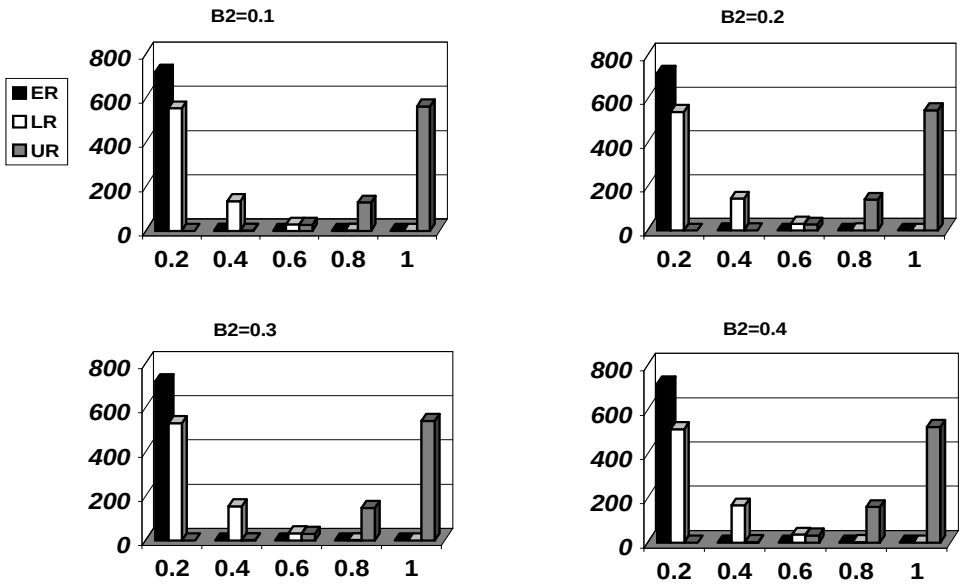

Figure 14: ircache SD trace tagging results, B1=5

object, the number of ER replicas is almost always very small, and UR replicas almost always dominate the whole population. The similarity of all 8 figures also suggest that this result is not very sensitive to the value of B1 and B2. Our tests on other ircache traces also support this observation.

6.1.1.2 Network Bandwidth

In this section we discuss tagging replicas according to their bandwidth constraints. Suppose the size of the object value we are considering is s_{obj}, The time it takes for replica R_i to deliver an new object value to replica R_j can be approximated by

$$L_{i,j} + \frac{s_{obj}}{B_{i,j}} + t_{proc} + t_{queue},$$

where $L_{i,j}$ is the communication latency between R_i and R_j, $B_{i,j}$ is the available communication bandwidth, t_{proc} is the processing time spent on both sides in order to finish the send and receive operations, and t_{queue} is the time spent on message queuing. With asynchronous dissemination, latency $L_{i,j}$ is generally ignored. Furthermore, we can assume

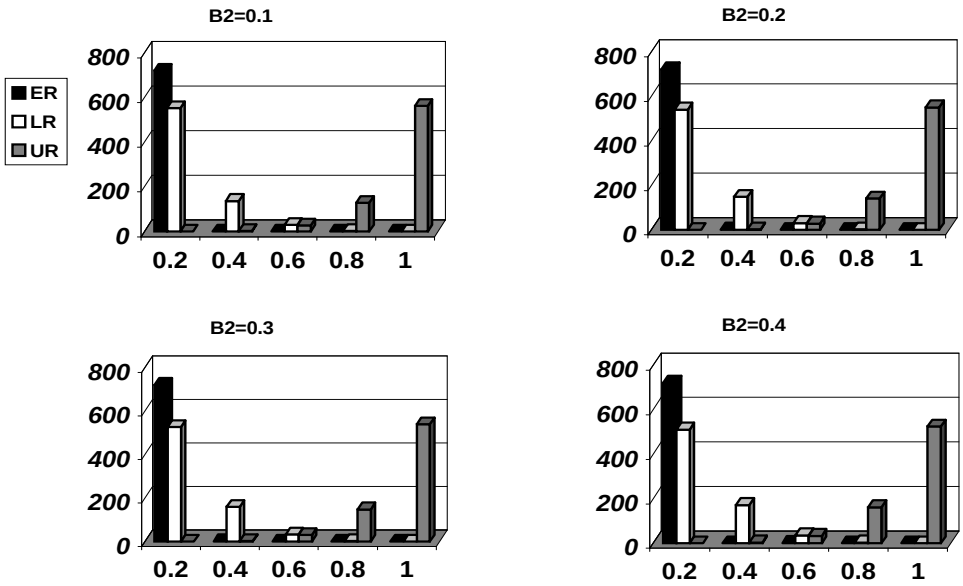

Figure 15: ircache SD trace tagging results, B1=20

that t_{proc}, t_{queue} are both small compared with $\frac{s_{obj}}{B_{i,j}}$. Therefore, the dissemination time of an object value from replica R_i to replica R_j is determined by $\frac{s_{obj}}{B_{i,j}}$.

Let k_j denote replica R_j's tolerance of "old" values, meaning that R_j needs to read a "fresh" value after it next k_j requests. So $\frac{k_j}{feq_j}$ denotes the time in which R_j needs an update after it is finished ($feq_j = \frac{n_j}{t}$, n_j and t are defined at the beginning of Section 6.1.1.1). We also know that $\frac{s_{obj}}{B_{i,j}}$ denotes how long it takes for replica R_i to disseminate an update to R_j. Therefore, if $\frac{s_{obj}}{B_{i,j}} < \frac{k_j}{feq_j}$ is satisfied, replica R_j will receive an update from R_i within its next k_j reads.

Suppose $target(R_i)$ denotes a set of replicas, which receive their updates from R_i. In other words, $target(R_i)$ is the set of replicas that need to receive update values via a "push" by R_i. We can then tag replicas according to its bandwidth resource as shown in figure 16. If we assume all k_js are equal to k, the condition in the algorithm can be simplified as $\frac{s_{obj}}{B_{i,j}} < \frac{k}{feq_j}$

Since LR replicas do not read their shared objects frequently, and UR replicas do not need to be "push"ed with new object values, $target(R_i)$ contains all ER replicas and a

```
Tagging_B()
  for every replica R_j ∈ target(R_i) do
      if (!(s_obj / B_i,j < k_j / freq_j)) then
          tag R_i as NB; //Narrow Bandwidth
          break;
  tag R_i as BB; //Broad Bandwidth
```

Figure 16: Replica Tagging According to Bandwidth Resource

subset of LR replicas. Suppose there are f members in LR that belongs to $target(R_i)$, we call this f the fan-out factor. It is a pre-defined value of the system. Therefore, $target(R_i) = ER \cup SLR_i$, where $SLR_i \subset LR$ and $|SLR_i| = f$.

Clearly, ER, LR and UR are disjoint sets and so are BB and NB. The group name can be used to denote replica tag. In other words, at any given time, a replica has one and only one of the following 6 tags: (ER, BB), (LR, BB), (UR, BB), (ER, NB), (LR, NB), and (UR, NB). It can change from one tag to another when the system conditions (read frequency and bandwidth resource) observed changed.

6.2 Algorithm

We present the algorithms for consistently disseminating newly generated values of replicated objects in this section. In particular, we first describe an adaptive object value dissemination protocol. It can be used as a transport layer support for different consistency protocols. At this time we support causal consistency in our system. Part of the future work is to implement additional consistency protocols on a common transport layer that supports adaptive dissemination. We assume that each replica R_i knows the tag of all other replicas (i.e. whether they are in ER, LR, or UR; BB or NB) when presenting our protocol. We then give a brief analysis of our dissemination algorithm. After that, we present our implementation of a causal consistency protocol. And finally, we relax the assumption of global replica tag knowledge and discuss how replica tags are maintained in the system.

6.2.1 Adaptive Update Dissemination

We use three techniques: direct send, rumor spreading and invalidation/pull. Direct send and rumor spreading are used to propagate update to eager readers and lazy readers in order to better satisfy their regular read requests. Invalidation is used to notify a large population of unpredictable readers that new object values are available. When they are needed, the new values will be pulled from others in the system.

The algorithm we present is fully distributed. Each replica in the system executes the same program. This adaptive update dissemination algorithm has two phases, "Push" and "Pull". The "Push" phase is initiated when upper layer sends an update (usually the result of a write operation). The "Pull" phase is initiated when a local invalidated copy is requested. Often the "pull" phase is initialized from the upper layer, i.e. the consistency layer. We will introduce the "pull" phase in our causal consistency implementation given in Figure 19

$R_i \in BB$ and initiates an update $< O, V, M >$
//O: object identifier; V: new object value; M: consistency control information
  R_i sends invalidate message to any $R_j \in UR$;
  R_i sends an update message$< O, V, M, target(R_i) - ER, 1 >$ to any $R_k \in target(R_i)$;

$R_i \in NB$ and initiates an update $< O, V, M >$
  R_i sends invalidate message to any $R_j \in UR$;
  if (BB is not empty)
    //let R_k be a random replica in BB
    R_i sends R_k an update delegation message$< O, V, M, target(R_i) - ER, 1 >$;
  else *// no BB replica is online, turn to best effort sending*
    R_i sends update message$< O, V, M, target(R_i) - ER, 1 >$ to any $R_p \in target(R_i)$;

$R_i \in BB$ and receives an update delegation message $< O, V, M, OLR, 1 >$
  R_i sends an update message$< O, V, M, OLR, 1 >$ to any $R_k \in OLR \cup ER$;

Figure 17: R_i initiates the update

Figure 17 shows the the protocol of when replica R_i initiates the update process. Serving a write request at a replica R_i triggers the update dissemination. The writer R_i initiates the propagation in one of the two cases:

- $R_i \in BB$: R_i sends the new object value to all replicas in $target(R_i)$.
- $R_i \in NB$: R_i tries to find a replica in BB group to delegate the dissemination. If no

```
R_i ∈ ER and received message< O,V,M,OLR,t >
//OLR: a partial list of receivers from round t; t: round number
  R_i delivers the < O,V,M >;

R_i ∈ LR and received message< O,V,M,OLR,t >
  if (< O,V,M > has not been processed)
      R_i delivers the < O,V,M >;
      R_i randomly picks a set NLR, such that NLR ⊂ LR and |NLR| = m_i;
      With probability PF_i(t), R_i sends < O,V,M,OLR ∪ NLR,t+1 > to NLR-OLR;
      Mark < O,V,M > as processed;
```

Figure 18: R_i receives and propagates the update

such replica is found (i.e. no BB replica is online), R_i sends the new object value to all replicas in $target(R_i)$ [2].

In both cases, R_i invalidates the UR set.

We need to point out that a certain BB replica may become heavily burdened by serving delegated updates of many NB replicas. The resource monitor should observe a bandwidth resource decrease by the aggregated consumption, and this may trigger the replica changes its tag from BB to NB. It does not affect the correctness of the algorithm.

For any LR replica R_j that receives the update, it further propagates the new value to m_j randomly chosen LR replicas with probability $PF_j(t)$. $PF_j(t)$ is computed based on the round number t field ([27]) in the update message so that it approaches zero when round number becomes large. Eventually the propagation process will stop. Figure 18 shows the propagation among LR replicas. This protocol is a slightly changed version of the propagation algorithm in [27]. It is based on rumor spreading technique, [28], [16]. Recent study in [27] shows that it is highly scalable and can tolerate large offline populations.

For replicas in ER or LR group, the local object values are kept "fresh" by receiving actively propagated updates. Read operations from upper layer are satisfied by their local values. For an UR replica, valid local values are used to satisfy upper layer reads. If local values are invalidated, it pulls a "fresh" copy from others.

[2] Please note that it is now a "best-effort" dissemination because it takes relatively long time for NB replica R_i to send to the whole $target(R_i)$ group.

6.2.2 Analysis

We give a brief analysis of the performance of our algorithm in this section. Our goal is to disseminate updates to a large group of interested peers, while being sensitive to application and resource heterogeneity, in order to better satisfy the application needs and efficiently utilize the network resources.

We use three techniques: direct send, rumor spreading and invalidation/pull, in our algorithm. Direct send and invalidation are mainly local effort, in that the local peer takes care of the sending task. When the sending is over, the task is finished. Rumor spreading, however, is a group effort, in that local peer only participates in sending out the update and when local sending is over, the task can still go on at other peers.

We are mainly concerned about the timeliness of an update's dissemination. In particular, these two bounds are of interest: local time bound on sending out a message(T_{local}), and group time bound on receiving the message (T_{group}). T_{local} is used to capture the local effort that is required from each peer in our system, while T_{group} is used to capture the effectiveness of our algorithm (i.e. how fast the updates can be propagated).

6.2.2.1 Local Time Bound

Suppose bw_{ij} is the bandwidth dedicated to our algorithm to communicate between replica R_i and R_j. The time it takes for R_i to send out one update message to R_j is $\frac{MSG_U}{bw_{ij}}$, where MSG_U denotes the size of the update message. A BB replica in figure 17 is the replica that sends out most update messages (assuming all m_is in figure 18 are less than or equal f) in our algorithm. Its update sending takes

$$t_U = \sum_{R_j \in target(R_i)} (\frac{MSG_U}{bw_{ij}}) = \sum_{R_j \in ER} (\frac{MSG_U}{bw_{ij}}) + \sum_{R_j \in SLR_i} (\frac{MSG_U}{bw_{ij}})$$

and its invalidation sending takes

$$t_I = \sum_{R_j \in UR} (\frac{MSG_I}{bw_{ij}})$$

where MSG_I denotes the size of an invalidation message.

The local effort of BB replica is bounded by $t_U + t_I$. And a NB replica, if cannot find any BB replica to delegate the update sending, also takes $t_U + t_I$ time to sends out messages. Therefore, the local effort of replicas in our system is bounded by $T_{local} = t_U + t_I$. Compared with a broadcast based algorithm, whose

$$t_U = \sum_{R_j \in ER+LR} (\frac{MSG_U}{bw_{ij}})$$

and

$$t_I = \sum_{R_j \in UR} (\frac{MSG_I}{bw_{ij}})$$

Thus, our algorithm has a lower local effort requirement, given $f \ll |LR|$.

6.2.2.2 Group Time Bound

The time it takes for an ER replica to get an update from others is decided by whether the initial sender is a BB replica (or whether the initial sender can find a BB replica to delegate the sending). If not, the algorithm turns into a best-effort sending process and there is no guarantee when this ER replica will receive the update. Otherwise, we learn from the definition of BB in Section 6.1 that an ER replica is guaranteed to receive the new value within its next k reads . In other words, T_{group} of ER is bounded by the time of k reads.

The timeliness of rumor spreading technique is usually measured by the number of rounds it takes for the message to be propagated to a certain percentage of the nodes. A detailed study can be found in [27] and [28]. For simplicity, we assume all the m_is in figure 18 equal f and no replicas are propagated with duplicate messages. In this case, T_{group} of LR is bounded by the time of $\log_f |LR|$ rounds.

There is no group time bound on UR, because each of them may get invalidations from either ER or LR replicas when an update is issued.

6.2.3 Upper Layer Consistency Protocol Example – Causal Consistency

In this section, we present a consistency protocol – causal consistency as an example of the multiple upper layer consistency protocols our algorithm can support. The algorithm presented in figure 19 is based on its vector clock implementation.

The causal consistency of ER and LR groups are ensured by the guaranteed delivery of all the updates, [6]. And for UR replicas, they pull new values from others. The vector clock ensures that no old values can be applied. Therefore, although staled values are possibly returned at UR replicas, values that violate causal consistency will not.

```
init()
  //initializing the meta data
  for j = 1 to N do
     t[j] = 0; //initialize the timestamp
  oqueue = ∅;
  iqueue = ∅;

read(x)
  //read the value of object replica x
  if(x ∈ ER ∪ LR)
     if (x is READABLE) return O[x];
     else throw READ_EXCEPTION;
  if((x ∈ UR) AND (x != ⊥)) return O[x];
  //last_writer is recorded during the invalidation phase
  <v, s> = last_writer.value_pull(x);
  for any ER and LR replica z do
     if (∃j ≠ i:s_z[j]>t_z[j])
        set z as NOT READABLE;
        set condition(z)=s_z[j];
  if (s_x[i]>t_x[i])
     t_x[i]=s_x[i];
  O[x] = v;
  return O[x];

write(x, v)
 //write x with new value v
 t[i] = t[i] + 1;
 O[x] = v;
 for every process p that holds an UR copy of x, enqueue(oqueue, < x, ⊥, i, t >);
 enqueue(oqueue, < x, v, i, t >);

value_pull(x)
 //return the value of x, together with clock
 return <O[x], t>;

applyd()
  //applying daemon, keeps running forever
  if (iqueue ≠ ∅)
     //let < x, v, j, s > = head(iqueue)
     if ((∀k ≠ j:s[k]≤t[k]) AND (s[j]=t[j]+1))
        dequeue(iqueue);
        t[j] = s[j];
        x = v;
        if ((x is NOT READABLE) and (condition(x)== s_x[j]))
           set x as READABLE;
```

Figure 19: Causal Consistency Implementation

6.2.4 Replica Tag Dissemination

In Section 6.2.1, we assume replica tags are known to all replicas in the system. In this section, we discuss how to maintain replica tag information in our system.

Each replica maintains two tag variables: F_r and B, where $F_r \in \{ER, LR, UR\}$ and $B \in \{BB, NB\}$. The algorithm described in Section 6.2.1 uses these variables to determine the proper dissemination techniques.

As described in Section 6.1, F_r and B are determined by monitoring read request service history and bandwidth consumption respectively. We assume that replica's tag change happens much less frequently than read or write requests. In a stable system where a replica's F_r and B never change, replicas can exchange their tag information when they join the system. In this case, they have accurate global knowledge when executing the algorithm in figure 17 and 18.

Peers can join (e.g. go online) and leave (e.g. go offline) the system at any time. We introduce a discovery service running at each peer to keep track of current online peers in the system, using periodic heartbeat messages.

When the system is unstable and a replica's F_r and B are updated, the new tag needs to be disseminated to others. Since the size of such messages is very small (usually a few bytes), we can piggy-back the tag information to the heartbeat message. The discovery service delivers the tag information to the membership service, which maintains the global replica tag information.

We need to point out that although we assume all replica tags are global knowledge in our protocol presented in Section 6.2.1, the correctness of our protocol can actually be satisfied by assuming only global knowledge of the whole ER group and partial knowledge of LR, UR, BB and NB. The study in Section 6.1.1 shows that the cardinality of ER group in real system is usually very small. Therefore, to keep this global knowledge in the system is unlikely to have big impact on the performance of our algorithm.

6.3 Summary

In this chapter, we introduced our agile dissemination technique, which combines direct send, rumor spreading and invalidation/pull to better facilitate system resources to meet different application needs. We gave the replica tagging algorithm which our dissemination technique is built upon. After that, we introduced the implementation of agile dissemination. Finally, we showed that the agile dissemination technique can be used to support different consistency protocols, and we gave as an implementation of causal consistency protocol as an example.

CHAPTER VII

IMPLEMENTATION AND EVALUATION

7.1 Mixed Consistency Model Implementation

We have implemented our mixed consistency protocol on top of the event delivery system Echo [31] [33] [32]. Echo is an event delivery middleware system developed at Georgia Tech. It was originally designed as a type of publisher-subscriber based system, featuring fast delivery of scientific computation data. It can efficiently transport large amounts of data by minimizing data copying and avoiding overly perturbing application execution [31]. In this implementation, we believe Echo is a good communication mechanism for our mixed consistency protocol to be built upon because 1) Echo supports heterogeneous platforms where, for example, the event data (e.g. a "long" value) can be interpreted differently at senders and receivers. 2) Echo supports FIFO reliable delivery of events by using internal queues and TCP. Based on the above advantages, we choose to use Echo as the transport layer implementation to implement our mixed consistency protocol.

Typed Event Channels: Echo provides an abstract communication mechanism called "channel", which supports multiple senders (also called "source") and multiple receivers (also called "sink"). Therefore, it is flexible to be used in different ways, such as one-to-one, one-to-many, or many-to-many communication mechanism in practice.

Our mixed consistency protocol assumes a one-to-one communication model between processes. Therefore, each process will maintain an "input" channel (that is, the process subscribes itself as the "sink" of the channel) in our implementation. We also choose the "typed" channels in order to considerably reduce the complexity in dealing with heterogeneous environment (See section 3 of [31]). The restrictions of using "typed" channels are that we cannot submit different event type data in the same channel. We overcome this problem by using the event data as a buffer of bytes and employing marshal and unmarshal procedures to give the buffer different interpretation.

System Architecture: The system architecture we designed at each node to implement our mixed consistency protocol is shown in Figure 20. The application interface of our

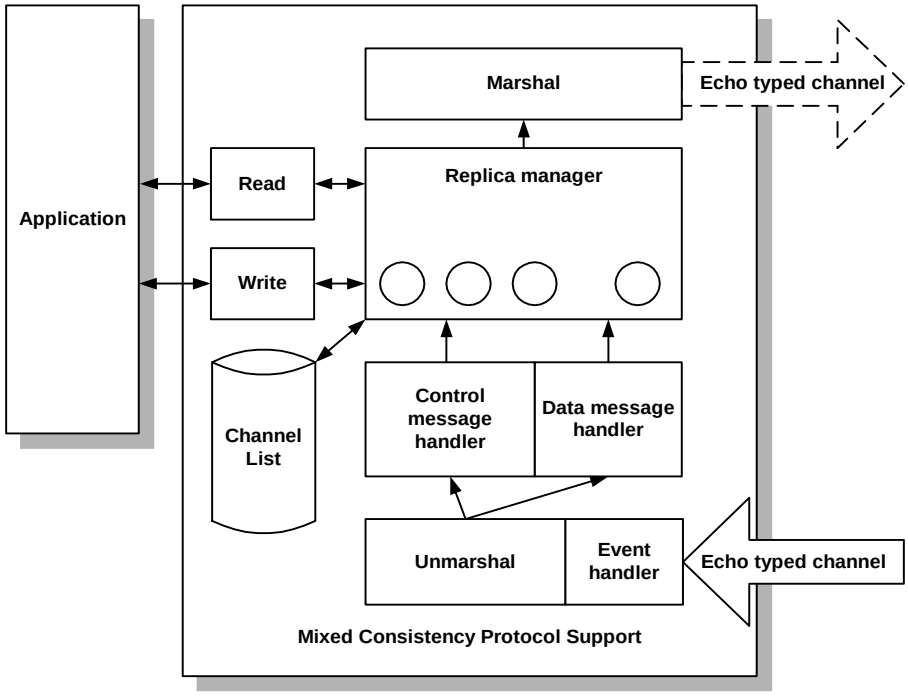

Figure 20: Mixed Consistency Implementation Architecture

mixed consistency protocol consists of only two operations, namely *read* and *write*. The protocol executes in a peer-to-peer fashion, where one echo typed channel is used as a one-way "input" communication mechanism. Figure 20 shows the internal software structure and its interactions of a typical process. In this figure, the *echo channel* at the bottom is the one established and maintained by the receiver (the current process shown in the figure), the one at the top is the "input" channel of the destination process (not shown in the figure). The arriving event (message) will first trigger the *event handler* to examine the event. An *unmarshal* procedure is then called to "open" the message and check the type of message. After that, different messages will be handed over to either *data message handler* or *control message handler*, which will in turn interacts with the *replica manager*. The *replica manager* is the core of the mixed consistency protocol, which maintains the

semantics of SC and CC replicas and responds to both application invocations (read/write requests) and communications from other processes (events received from the channel). It will send out messages through a *marshal* procedure to generate a format that is consistent with the echo typed channel definition. Each process also maintains a *channel list*, which is a cached copy of all available channels in the system. Its purpose will be discussed later.

Channel management: Suppose there are N processes in the system, we will have N input channels to be created and maintained because each process subscribes itself as a "sink" of one distinct channel. When a process p sends a message to process q, the sender p will first subscribe itself as a "source" of q's input channel, and then sends the message by submitting the event to it. Echo channel encapsulates all transportation parameters like IP address and port number and provides a unique identifier (channel ID) for accessing the channel. To find the destination process's channel ID is the first step of being able to communicate. We design a *channel manager* in the system, which maintains a list of all available channels of each process. The *channel manager* itself maintains a public "well-known" channel for each process to register its own channels with it. Each process caches a local copy of the entire channel list. So when sending a message, the sender process does not have to contact the channel manager to obtain the destination channel ID. Because a process is unlikely to change its channel ID during execution in our implementation, the local cache copy of channel list is effectively "read-only" to each processes. Therefore, we do not have a consistency problem in maintaining channel list cache.

7.2 System Performance Evaluation

Criteria: We have developed the mixed consistency model to meet the data sharing needs of heterogeneous users. Based on the mixed consistency model, we developed a protocol that combines existing SC and CC protocols together to implement a set of mixed consistency interfaces. We believe that by combining both SC and CC together our system can provide greater flexibility to the application so that the requests of heterogeneous users can be serviced within reasonable time constraints under varying resource limitations. Therefore, the metric of interest that we are focusing on is *average invocation time* that the application

experiences when invoking a particular *read* or *write* request.

Challenge: We conducted a variety of experiments to measure the above metric for different values of system parameters. The challenge of measuring the performance of our mixed consistency is that there are many parameters that can potentially impact the application response time. We divide them into the following categories:

- **system size parameters**: number of processes, number of objects, object size.
- **system composition parameters**: process tag (i.e. what are the CC processes and what are the SC processes), replica tag (i.e. what are the CC replicas and what are the SC replicas).
- **trace composition parameters**: read operation percentage, write operation percentage. We feed each process with randomly generated traces. By adjusting the read and write operation percentage, we can measure the performance of different processes in a "reader" (where read operation dominates the trace), "writer" (where write operation dominates the trace), or "neutral" (where neither read or write dominates the trace) role.
- **failure parameters**: failure possibility, recovery possibility (we tuned this parameter by adjusting the upgrade and downgrade operation percentage in the trace).

Each combination of those parameter values may present a different system running condition, thus may lead to very different system performance behavior. In order to accurately capture those system behaviors, we adjusted different parameters in a series of experiments. The result presented in this thesis is computed based on the average invocation time of 10 runs, and the unit of measurement is microsecond (10^{-6} second).

Platform: We conducted our experiments on two different platforms. First, we ran benchmark tests and trace driven tests on CoC "warp" cluster, which consists of 56 nodes (warp[1..56]). Each node is equipped with dual x86 3GHz processors and 1GB RAM. They are interconnected via Gigabit (1000 Mbps) Ethernet. These machines are running Linux

(kernel version 2.6.9-34.ELsmp). The Echo we used is version 2.2. We then ran several trace driven tests on Emulab [34] with two different network topology settings. The detail of Emulab experiments will be discussed in Section 7.2.3.

7.2.1 System Evaluation through Micro Benchmark

We first measure the cost associated with read and write operations in basic system settings. The purpose of these experiments is to measure the baseline overhead of our mixed consistency protocol experienced by the upper layer application. Our MC protocol is built by combining both CC and SC protocols together. It is reasonable that we examine the their baseline performance first. We then use a different mixture of processes, and objects to measure the baseline of our MC protocol.

Experiment 1: CC protocol benchmark We first conduct a series of tests to examine how the number of processes can impact the response time of CC read and write operation of our mixed consistency protocol. We force the system to execute in CC mode by setting all processes to be CC, and share among them one CC object with size 64 bytes. Each process executes a randomly generated read and write sequences on this object. We measure the response time based on the average of at least 10 read/writes. The result is shown in Table 5 (also plotted in Figure 21). Please note that we measure the time of read operation in two different ways: the row of *read* measures the "original" read operation where the actual value is copied from Replica Manager to the application, while the row of $read^*$ measures an "optimized" version of read operation where only the pointer to the value is returned. We will use this notion throughout the rest of performance evaluation.

Table 5: CC read/write benchmark with increasing process number, in microseconds

number of process	2	12	22	32	42	52
$read^*$	0.6	0.6	0.7	0.6	0.8	0.7
read	8.8	8.3	8.6	9.4	9.2	9.5
write	22.3	19.3	24.1	17.8	22.4	24.4

From Table 5, we know that the response time of CC read and CC write does not increase with the number of processes in the system. This is consistent with the fact that

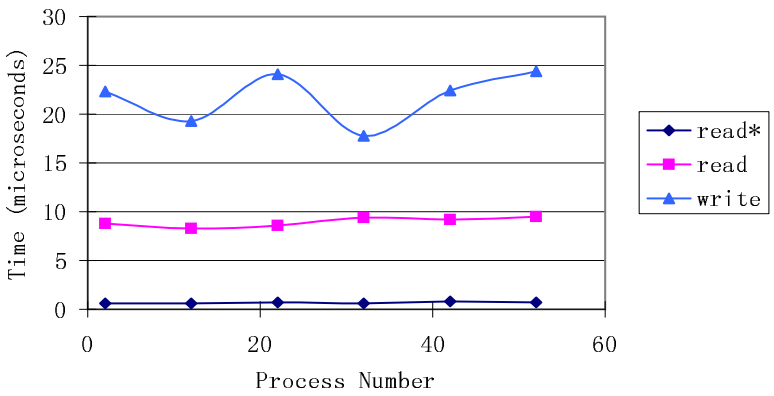

Figure 21: CC read/write benchmark with increasing process number

in CC protocol, both read and write only incur local computation cost. Even though the matrix clock size increases with the number of processes, the dissemination time is not included into our performance metrics because dissemination happens in the background and is performed in parallel with the application. We also notice that the write operation takes longer to finish than the read operation. This is because the write operation needs to perform extra operations in order to prepare the necessary data structure (i.e. Echo event) and connect with Echo.

We then examine the impact of object size on the response time. We chose a setting of 5 CC processes to conduct this test. From the result of Table 5, we know that to choose a different process number won't reveal more interesting information. We vary the size of the CC object each process share. The result is shown in Table 6 (also plotted in Figure 22)

Table 6: CC read/write benchmark with increasing object size, in microseconds

size of object (bytes)	32	64	128	256	512	1024
*read**	0.8	1.0	0.8	0.7	0.9	0.9
read	8.7	8.5	10.3	11.9	11.6	13.5
write	24.1	22.8	25.3	27.6	26.5	29.6

From Table 6, we found that the response time of CC read and CC write does not increase significantly with the size of objects. We originally thought that the result would be different because the local memory access time increases for both read and write when

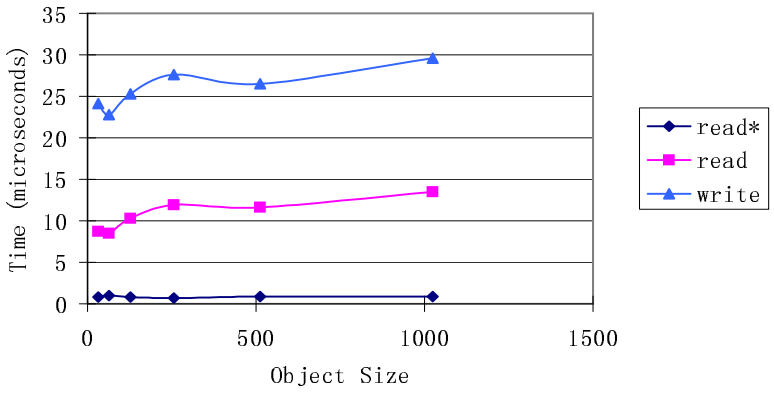

Figure 22: CC read/write benchmark with increasing object size

the object size increases. It is however understandable because the local memory access time is relatively small compared with the actual execution cost of the protocol. Thus it is not shown as a significant factor in the result.

In a third test, we examine the impact of number of objects on the response time. We conducted the test in a setting of 5 processes with fixed size object (64 bytes). We vary the number of objects (namely 1, 5, 10, 15, 20, 25, 30) shared by those processes and measure the response time of both CC read and write. We used fixed process number and object size in this test because we do not believe that varying those parameters will reveal more interesting results size than those we have already explored in Tables 5 and 6. The result is shown in Table 7 (also plotted in Figure 23).

Table 7: CC read/write benchmark with increasing object number, in microseconds

number of object	1	5	10	15	20	25	30
read*	0.7	0.6	0.8	0.8	0.6	0.9	0.6
read	8.3	8.2	8.9	8.8	9.5	8.8	9.2
write	22.3	23.6	24.8	23.4	27.6	24.2	25.6

From Table 7, we observe that the response time of CC read and write does not increase with the number of objects. The result is consistent with our expectation in that even though the matrix clock size increases with the number of objects, the dissemination time is not included into our performance measurements because the actual dissemination

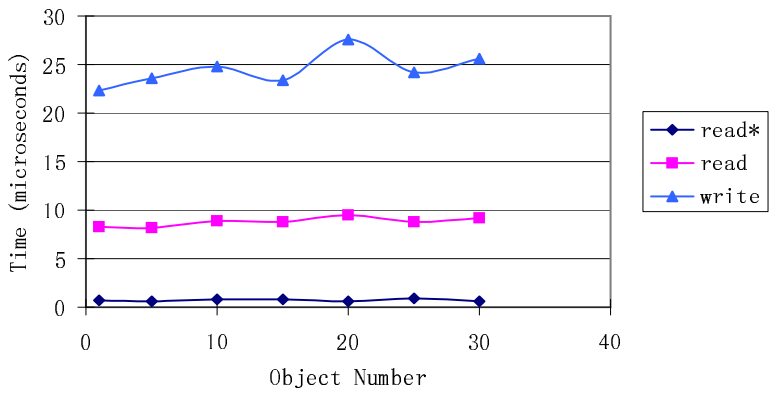

Figure 23: CC read/write benchmark with increasing object number

happens in the background and is executed in parallel with the application.

Experiment 2: SC protocol benchmark In the first test of this experiment, we examine how the number of processes can impact the response time of SC read and write operation of our mixed consistency protocol. We force the system to execute in SC mode by setting all processes to be SC, and share among them one SC object with size 64 bytes. Each process executes a randomly generated read and write sequence on this object. We measure the response time based on the average of 10 reads/writes. The result is shown in Table 8 (also plotted in Figure 24).

We realize that since both SC read and write operation can potentially involve token request communication with homenode, the response time will increase. Therefore, we decided to measure their average response time separately in order to better understand where the cost comes from. In Table 8, the data in the read (write) row in the table measures the average response time where there is no token request (i.e. process locally cached token has not been revoked), while the data in the $read^1$ ($write^1$) row measures the average response time where there is token request. We will use this notion throughout the rest of Experiment 2.

From Table 8, we observe that:

1. The response time of SC read and write does not increase when there are no token

Table 8: SC read/write benchmark with increasing process number, in microseconds

number of process	2	12	22	32	42	52
*read**	1.3	1.2	1.1	1.1	1.2	1.7
read	10.4	9.8	11.2	11.5	10.4	12.7
read[1]	3765.4	3998.2	4016.5	4023.5	4285.3	4396.6
write	77.5	73.8	74.2	72.5	79.3	78.6
write[1]	1987.3	2123.3	2224.5	2173.5	2533.0	2635.3

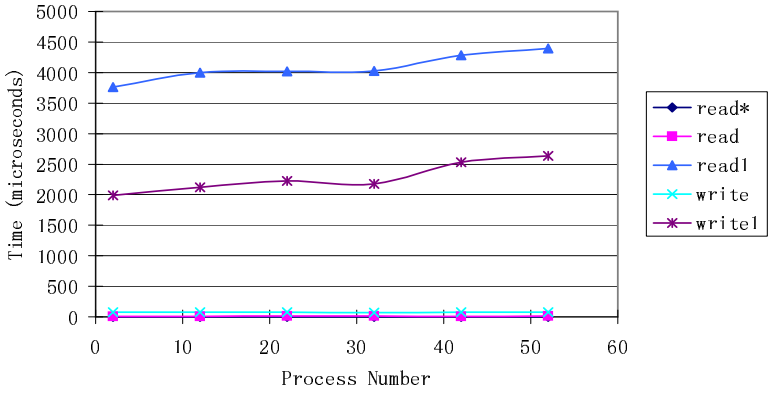

Figure 24: SC read/write benchmark with increasing process number

requests involved. This is reasonable because SC protocol only incurs local cost when the cached token is valid.

2. The response time increases for both SC read and write when a token is needed. It is consistent with our implementation in that we do not keep track of each individual token allocation in the homenode. When a new token request comes in and the homenode decides to revoke the outstanding token, it sends the revocation message to all processes in the system. Therefore, the token request response time increases when the system size increases.

3. When there are token requests, the response time of SC read is actually slower than SC write in the same system setting. It is reasonable because when a read token is obtained, the SC protocol needs to ask for the latest writer to pull the value. It incurs extra network communication cost.

We then examine the impact of object size on the response time. We chose a setting of

5 SC processes to conduct this test. We vary the size of the SC object each process shares. The result is shown in Table 9 (also plotted in Figure 25).

Table 9: SC read/write benchmark with increasing object size, in microseconds

size of object (bytes)	32	64	128	256	512	1024
$read^*$	1.2	0.9	1.1	1.1	1.3	1.4
$read$	12.7	10.1	12.6	12.5	15.3	16.8
$read^1$	3799.3	3666.4	3824.2	3849.2	3814.8	3866.5
$write$	89.2	82.2	85.6	84.4	87.1	85.4
$write^1$	2024.5	2122.3	2089.7	2216.5	2119.3	2186.2

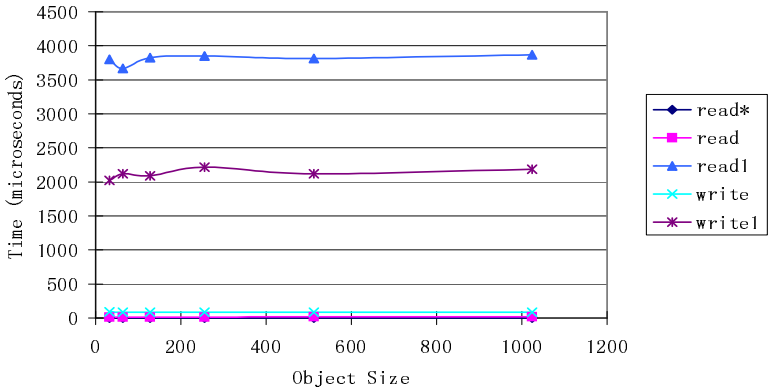

Figure 25: SC read/write benchmark with increasing object size

From Table 9, we observe that:

1. The response time of SC read and write does not increase significantly when there are no token requests involved. This is reasonable because the SC protocol only incurs local cost when the cached token is valid, even though the local memory access time should increase with object size, its impact is very small.

2. The response time increases for SC read when a token is needed, although from 32 to 64, and 256 to 512 are two exceptions. It is a reasonable result since SC read involves pulling the most recent value from the latest writer when a new read token is obtained. The network transmission time increases when the object value size increases. We believe the exceptions are caused by network jittering and/or temporary system load hike.

3. The response time does not increase with object size for SC write even when a token is needed. It is understandable because the increase of object size only affects its local write time and can be omitted.

In a third test, we examine the impact of number of objects on the response time. We conducted the test in a setting of 5 processes with fixed size object (64 bytes). We vary the number of objects (namely 1, 5, 10, 15, 20, 25, 30) shared by those processes and measure the response time of both SC read and write. We used fixed number of processes and object size in this test because we do not believe that varying those parameters will reveal more interesting results than those we have already explored in Tables 8 and 9. The result is shown in Table 10 (also plotted in Figure 26).

Table 10: SC read/write benchmark with increasing object number, in microseconds

number of object	1	5	10	15	20	25	30
*read**	1.1	1.3	1.1	1.1	1.4	1.2	1.3
read	10.2	11.3	9.6	11.2	14.2	12.1	12.6
$read^1$	4001.2	3982.1	4212.7	4133.5	4238.4	4024.8	3987.2
write	80.1	79.2	78.8	80.2	78.2	81.6	81.2
$write^1$	1987.3	1998.6	1897.2	2020.8	1921.4	1943.3	1977.9

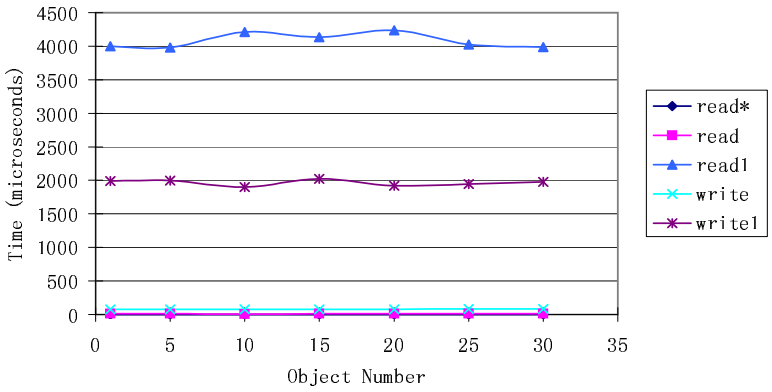

Figure 26: SC read/write benchmark with increasing object number, in microseconds

From Table 10, we observe the response time of SC read and write operations does not increase with the number of objects. It is reasonable because even though the bigger object size means increased matrix clock size, the increase is minimum and does not have

an appreciable effect on time.

7.2.2 System Evaluation through Synthetic Workloads

Locality of access determines which replicas to be used and it depends on the behavior of the applications. Therefore, it is desirable to evaluate the system using traces collected when such applications are deployed for actual use. However, currently available traces we are aware of mostly come from distributed file system and the world-wide web domains. The read/write sharing patterns for these are very different from the distributed object sharing applications. Furthermore, it is common in these applications that there is only a single writer for a given piece of data. Therefore, we choose not to use those traces to evaluate our system. Instead, we used synthetically generated traces based on important parameters (i.e. trace composition and failure parameter) to evaluate our system.

Trace Generation and Format: The trace file consists of a list of operations. The process running the trace executes those operations in sequential order. There are four possible operations in the trace. They are:

- READ: the format of the operation is {READ, {object ID}}. The process executing this read operation will issue a read request to our replica manager, which in turn executes the mixed consistency protocol based on the process and object tag involved, and returns the result to the process. For example, {READ 0} stands for "read the value of object 0".

- WRITE: the format is {WRITE {object ID} [{object value}]}. For example, {WRITE 1 132} stands for "write the value of object 1 to 132" (assuming object 1 is an integer object). Since our mixed consistency system supports multiple types of object values (for example, both integer and string objects can be shared), it is difficult to specify the value of the object in the trace file without knowing the type of {object ID}. However, since each writer knows what type of object it writes to, we can let the writer decide a value to write. Furthermore, our performance evaluation will not be impacted by the actual value that is being written. It is safe for the writer to

randomly generate value for {object ID}, as long as the size of the value is consistent with the object size in the experiment setting. Therefore, in our synthetic workloads evaluation, we omit the {object value} part in the trace and each process (writer) will generate a random value for the object it writes. For example, if we are running experiments on size 64 string objects, {WRITE 1} stands for "write a string of 64 bytes to object 1".

- DOWNGRADE: the format is {DOWNGRADE {object ID}}. The process executing this downgrade operation will issue a downgrade request to our replica manager, which in turn downgrades the replica from SC to CC. For example, {DOWNGRADE 2} stands for "downgrade object 2 from SC to CC".

- UPGRADE: the format is {UPGRADE {object ID}}. The process executing this upgrade operation will issue an upgrade request to our replica manager, which in turn upgrades the replica from CC to SC. For example, {UPGRADE 4} stands for "upgrade object 4 from CC to SC".

Please note that the {object ID} in each operation is randomly generated. Therefore, according to different system settings, it is possible that process 3 has a CC copy of object 0 but it is executing a trace containing a {DOWNGRADE 0} operation. Our replica manager expects that and will automatically omit operations that are inconsistent with the current object tag and/or process tag and move on. It does not affect the correctness of our protocol. Another similar example is that a SC process executes a READ operation on a CC replica, or a CC process executes a WRITE on a SC replica. Normally the mixed consistency protocol will throw an exception because they violate the access constraints. However, in order to complete our experiments without disruption, we disabled the exception in our workloads experiments. The protocol will simply ignore the violating command and continue with the next operation.

We use a combination of four integers to identify each workload. For example, a (50, 40, 6, 4) workload means that in this randomly generated workload, there are 50% read operations, 40% write operations, 6% downgrade operations, and 4% update operations.

For instance, if the trace file size is 400 (meaning there are 400 operations in the trace file), 200 of them are read, 160 are write, 24 are downgrade, and 16 are upgrade operations.

System Setting and Key Scenarios: We conduct a series of trace driven experiments, in order to measure the performance of mixed consistency protocol under different scenarios. We first chose a system setting of 20 processes, where 5 are SC processes and 15 are CC processes. Each process maintains 5 objects with size 64 bytes. SC process maintains SC replicas, and CC process maintains CC replicas. We believe this system setting actually represents a typical setting where our mixed consistency protocol would be used, i.e. there are a handful resourceful nodes that are capable of maintaining strong consistency among replicas, where a majority of nodes are maintaining relaxed consistency (CC) to achieve flexibility. Under this system setting, we run tests in six different scenarios. They are:

- **Scenario 1: Reader dominant system, failure-free** We use trace (75, 25, 0, 0) in this scenario so that a majority of the operations each process executes is read. This scenario represents the information sharing applications in real world (for example, stock quote widget) where participants consume the information much more frequently than generating new information.

- **Scenario 2: Writer dominant system, failure-free** We use trace (25, 75, 0, 0) in this scenario so that a majority of the operations each process executes is write. This scenario represents a set of information sharing applications in real world (for example, software sensors) where participants generate the information much more frequently than consuming it.

- **Scenario 3: Read/Write balanced system, failure-free** We use trace (50, 50, 0, 0) in this scenario so that each process executes an equal number of read and write operations. This scenario represents a set of information sharing applications in real world (for example, environment object in an interactive game) where participants both generate and consume the information frequently.

- **Scenario 4: Reader dominant system, w/ failure** We introduced downgrade and

upgrade in the trace to measure the performance in a setting similar to scenario 1 but with the possibility of failure and recovery. We assume the downgrade and upgrade happen relatively infrequently compared with read or write operations. In order to obtain comparable results with Scenario 1, we injected 5 downgrade operations and 5 upgrade operations in the same trace that was used in Scenario 1. Otherwise if we randomly generate different traces for Scenario 4 (say, a new (70, 20, 5, 5) trace), the results will be incomparable.

- **Scenario 5: Writer dominant system, w/ failure** We introduced downgrade and upgrade in the trace to measure the performance in a setting similar to scenario 2, with the possibility of failure and recovery. Similarly, we injected 5 downgrade operations and 5 upgrade operations in the same trace that was used in Scenario 2.

- **Scenario 6: Read/Write balanced system, w/ failure** We introduced downgrade and upgrade in the trace to measure the performance in a setting similar to scenario 3, with the possibility of failure and recovery. Similarly, we injected 5 downgrade operations and 5 upgrade opreations in the same trace that was used in Scenario 3.

The results of the experiment are shown in Table 11. Since we have both CC and SC operations in the system, the application experiences different response time when invoking read/write operations against different replicas (i.e. CC or SC). We realize that the average response time of read/write operations (no matter CC or SC) can hardly capture the essential performance difference across scenarios because these operations are trace dependent. However, we think the average response time of read/write is still a reasonable performance indicator in that it gives a measurement of what is the response time the application can expect from the mixed consistency protocol when running in either reader/writer dominant or neutral environment regardless of a variety of other parameters, such as SC, CC tags.

Table 11: Trace driven experiments (5 SC, 15 CC), in microseconds

scenario	1	2	3	4	5	6
read	2195.6	4109.2	2278.4	2021.3	4002.7	2134.5
write	496.8	2122.4	893.5	677.5	2423.5	974.3

From Table 11, we can see that for similar problem settings (i.e. Scenario 1 and 4, 2 and 5, 3 and 6), the read/write response time are comparable. This is consistent with what we expected. Our mixed consistency protocol provided a new way of addressing failures in information sharing systems so that the computation can go on with the presence of failed (non-responsive) processes. We do not see any correlation among scenario 1, 2 and 3 (4, 5 and 6). That is understandable because if a particular trace contains many SC operations on the same object, the process will have to communicate more with the homenode in order to obtain a token for read and write. The average response time of read and write will inevitably increase. Therefore, the result is somewhat trace dependent, and we cannot compare among scenario 1, 2 and 3 (4, 5 and 6).

We then conduct two more trace driven experiments with different system compositions on the above six scenarios with the same trace file:

- Experiments in Table 12 shows a system with 10 SC processes and 10 CC processes.
- Experiments in Table 13 shows a system with 15 SC processes and 5 CC processes.

The combined results of Table 11, 12 and 13 are plotted in Figure 27.

Table 12: Trace driven experiments (10 SC, 10 CC), in microseconds

scenario	1	2	3	4	5	6
read	3012.7	5922.6	3345.2	3078.8	6102.5	3335.2
write	643.3	3301.2	1014.5	792.8	3508.4	1134.3

Table 13: Trace driven experiments (15 SC, 5 CC), in microseconds

scenario	1	2	3	4	5	6
read	3889.8	7900.5	4104.9	3657.8	8182.4	4002.4
write	812.2	3797.4	1135.6	911.4	4241.7	1233.9

From Figure 27, we can observe that for a particular scenario, the average response time of read and write increases when the system contains more SC processes (assuming other parameters remain unchanged). This is consistent with our expectation such that our MC protocol can provide great flexibility to the application. In particular, it can provide the application with different response time by dynamically adjusting the system composition,

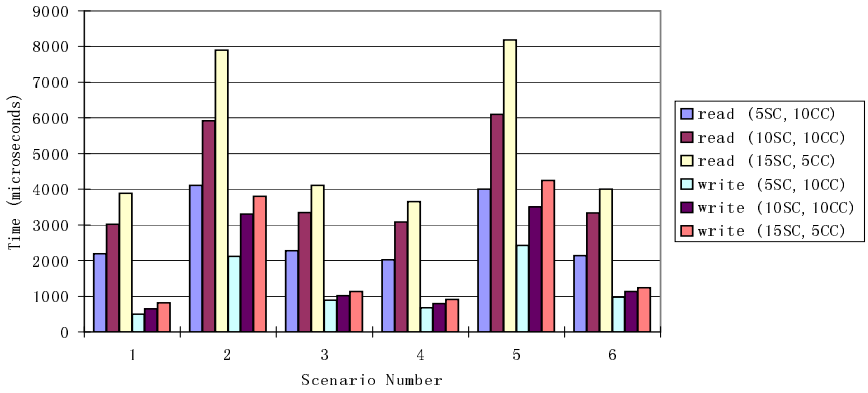

Figure 27: Combined results of trace driven experiments (CoC warp cluster)

i.e. SC and CC process numbers, while keeping the rest of system parameters relatively stable.

7.2.3 Emulab Experiments

We also conducted our experiments in Emulab environment. Emulab is an integrated set of network experimental environment, which provides simulated, emulated, and wide-area network testbeds ([34]). It consists of 328 PCs (including 160 3GHz Dell PowerEdge 2850s). A complete list of test hardware can be found in [66]. Emulab users can specify network topologies with a variety of parameters such as link bandwidth in order to start an experiment. Once the experiment is set and OS image is loaded, the user will have "root" access to those nodes dedicated to the experiment.

We set up two network topologies:

- Topology A (Figure 28): It consists of 6 nodes (Node[A-F]). Node A, B, and C are interconnected by a 100Mbps LAN switch. Node D, E, F are connected with A, B, and C by a 1.0 Mbps link (denoted by a dotted line in the figure), respectively. For example, Node A is connected with Node D via a 1.0 Mbps link. Every node is running Linux RedHat 9.0.

- Topology B (Figure 29): It consists of 21 nodes (Node[0-5], Node[1-5][1-3]). Node[0-5]

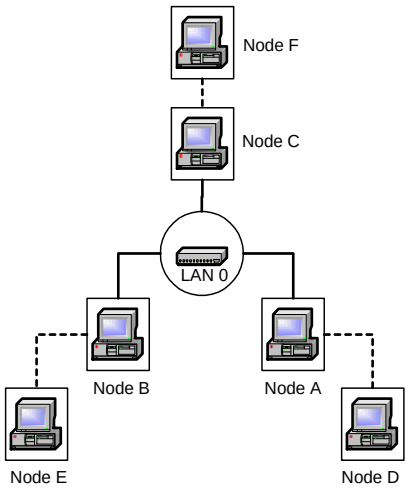

Figure 28: Emulab topology A (6 nodes)

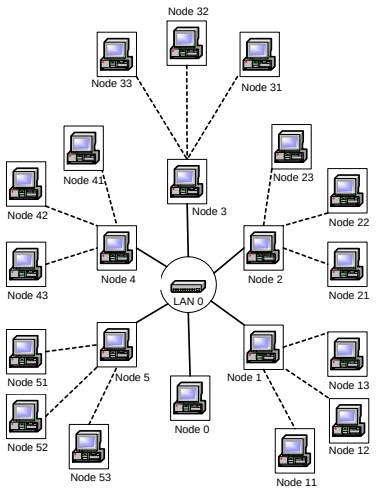

Figure 29: Emulab topology B (21 nodes).

are interconnected by a 100Mbps LAN switch. Node[x][1-3] ($1 \leq x \leq 5$) are connected with Node[x] by a 3.0Mbps link (denoted by a dotted line in the figure), respectively. For example, Node 21 is connected with Node 2 via a 3.0 Mbps link. Every node is running Linux RedHat 9.0.

Bandwidth Impact on Performance: We run a series of tests on Topology A, in order to expose the bandwidth impact on different MC protocol operations. We place two processes in different locations in this topology, and use randomly generated trace to drive the tests. Both process share 1 object with size 64 bytes. We dedicate Node A as the homenode. For simplicity, we assume both processes to be either SC or CC, and SC process maintains SC replica and CC process maintains CC replica. The result is shown in Table 14. In this table, each column represents a test scenario. For example, "SC (B,C)" stands for the scenario where one SC process is running at Node B, and another SC process is running at Node C. Rows of $read^1$ and $write^1$ denote the time where token operations happened (token request/revoke/reply/).

From Table 14, we can see that slow link (in scenario (B,E) and (E,F)) in the path will have significant impact on token SC operations. It does not have strong impact on non-token SC operations. It is consistent with our expectation in that token operations require additional messages to be transferred over the network, and a low bandwidth link

Table 14: Emulab test on bandwidth impact (Topology A), in microseconds

scenario	SC (B,C)	SC (B,E)	SC (E,F)	CC (B,C)	CC (B,E)	CC (E,F)
read	130.6	134.3	130.2	36.4	107.3	116.4
read[1]	6283.5	7912.7	7881.8	-	-	-
write	268.3	243.1	288.3	104.9	178.6	199.5
write[1]	2995.0	3329.8	3426.6	-	-	-

will have a negative impact on the performance.

Failure Rate Impact on Performance: We conduct a series of tests on Topology B to examine the failure rate impact of the MC protocol. We let all SC processes run on nodes with 100Mbps bandwidth, and all CC processes run on nodes with 3Mbps bandwidth. We dedicate Node 0 as the homenode for all the shared objects. Therefore, there are 5 SC processes (running on Node[1-5]) and 15 CC processes (running on Node[1-5][1-3]). We assume that there are 5 objects shared by all the processes. The object size is 64 bytes. For simplicity, we also assume SC processes maintain SC replicas and CC processes maintain CC replicas.

We run three different scenarios with varying failure parameters (0 failure, 5% failure, 10% failure):

- Scenario 1: Read dominant trace. Initially starts with (75, 25, 0, 0), we then introduce 5% failure in the trace (i.e.(70, 20, 5, 5)), and finally 10% failure (i.e. (65, 15, 10, 10)).
- Scenario 2: Read/write balanced trace. Initially starts with (50, 50, 0, 0), we then introduce 5% failure in the trace (i.e.(45, 45, 5, 5)), and finally 10% failure (i.e. (40, 40, 10, 10)).
- Scenario 3: Write dominant trace. Initially starts with (25, 75, 0, 0), we then introduce 5% failure in the trace (i.e.(20, 70, 5, 5)), and finally 10% failure (i.e. (15, 65, 10, 10)).

The results are shown in Table 15 (also plotted in Figure 30). From the results, we can see the following:

Table 15: Emulab test on failure rate impact (Topology B), in microseconds

scenario	1	2	3
read (0% failure)	5574.9	4741.8	2144.5
write(0% failure)	4686.3	3374.8	2774.3
read (5% failure)	7611.1	3778.1	3047.3
write(5% failure)	4292.3	2822.2	1985.1
read (10% failure)	9593.8	3003.2	3318.9
write(10% failure)	2122.2	4833.9	10953.0

1. The response time of both read and write operations decreases when failure rate increases. This is consistent with what we expected. Because when failure rate increases, more SC replicas will downgrade into CC replicas. This will lead to less token operations in the system. From previous experiments, we know that token operations are very expensive. Therefore, less token operations will lead to less average response time.

2. The response time increases when more read operations are executed. Because we used randomly generated trace to run experiments, many factors can lead to this result such as execution timing and trace contention. For example, trace X contains more read operations than trace Y. In trace X, many read operations are performed on the same SC object consecutively, which means the client can cache the read token when executing these operations. Therefore, it actually requires less token operations even if the read operation number is big. In trace Y, even when it's total read number is small, it is possible that more read operations are separated from each other by a write operation to that same object. Therefore, more token operations will be generated, which will lead to longer average response time.

7.3 Summary

In this chapter, we presented our design and implementation of the mixed consistency protocol, and conducted a series of experiments to measure the average response time experienced by upper layer applications. The performance data we obtained are mostly consistent with what we expected from our mixed consistency protocol. We identify many parameters that can potentially impact the system performance. The challenge is how to

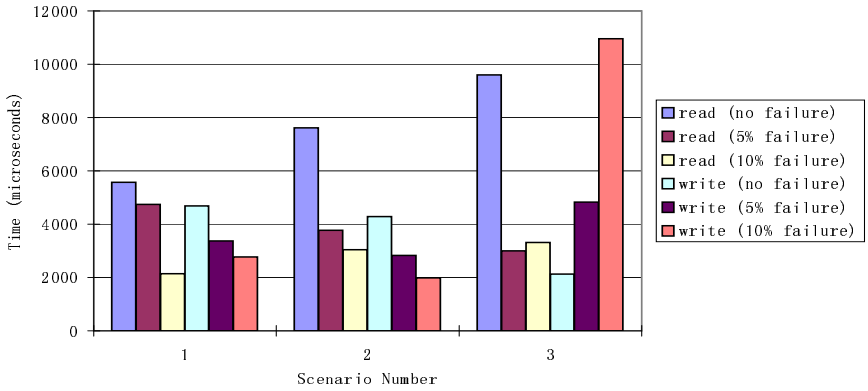

Figure 30: Emulab trace driven experiment result (topology 2)

obtain results for different combinations of those parameters. To run experiments for all combinations is theoretically doable but unnecessary because we are interested in identifying settings that can potentially be used in real world applications. Therefore, we chose to fix some parameters and vary the rest in different tests in order to expose the impact of the those parameters. The pros of this approach is that we were able to isolate the performance impact of those parameters from others. The cons is that we still had a lot of experiments to run, which prolonged the performance measurement period (some of the experiments are executed in different week). Since we do not have exclusive usage of the hardware platform (especially CoC warp cluster), we initially thought it would be hard to observe consistent performance data across different experiments. However, the results are fairly consistent across experiments. We figured that there are two possible reasons:

1. The warp cluster has plenty system resources, and during the time (12:00am - 3:00am EST) most of the experiments were run, the system load is relatively light.

2. Our mixed consistency protocol does not saturate the network communication, which makes it more resilient to changing system load (mainly network traffic).

Mixed consistency offers a tradeoff between consistency and performance. It allows both strong and weak replicas of the same object co-exist in the system at the same time, in order

to meet the heterogeneous challenge faced by each individual user. It also provides a innovative downgrade/upgrade mechanism to tolerate failures and changing levels of available system resource. This is validated by our experiments.

CHAPTER VIII

CONCLUSION AND FUTURE DIRECTIONS

With the rapid proliferation of mobile devices and wireless networks, a large number of new computing devices besides servers and PCs have been connected to the Internet. All together they can provide us the ability to access information from anywhere at any time. However, different user needs, varying computing power and communication capabilities of different devices have posed a serious challenge to information sharing applications because 1) for the same piece of information, different users may have different requirements for how it should be shared; 2) the resources that each device can contribute to such information sharing application vary, i.e. the resource abundant nodes can afford expensive protocol to ensure the consistency of the information shared while others are not able to do it. Therefore, how to enable information sharing across computing platforms with varying resources to meet different user demands is an interesting and important problem for distributed systems research.

In this thesis, we addressed the heterogeneity challenge faced by such information sharing systems. We assume that shared information is encapsulated in distributed objects, and we use object replication to increase the scalability and robustness of an information sharing system, which introduces consistency problems. Many consistency models have been proposed in recent years but they are either too strong and do not scale very well, or too weak to meet many users' requirements. Therefore, we believe a single level of consistency (like what is commonly done in existing systems) cannot meet the challenge of maintaining consistency in a heterogeneous environment. Instead, we propose a Mixed Consistency (MC) model as a solution. We introduce an access constraints based approach to combine both strong and weak consistency models together. As a result, our model allows both strong and weak replicas of the same object to coexist in the system at the same time. In order to utilize existing protocols of both strong and weak consistency models to implement the new

MC model, we propose a MC protocol that combines existing implementations together with minimum modifications. We also introduce a non-responsive process detector in order to tolerate crash failures and slow processes/communication links in the system. We also explore the possibility of addressing the heterogeneity challenge in the transportation layer by giving an agile dissemination protocol that can take into consideration both user needs and resource constraints when disseminating updates. We implement our MC protocol on top of a distributed publisher-subscriber middleware, Echo. We finally measure the performance of our MC implementation through a series of experiments designed to expose the impact of different system parameters. The results of the experiments are consistent with our expectations.

8.1 Contribution

The major contributions of this thesis are summarized below:

- We identify two important heterogeneity factors that should be considered by large information sharing systems, namely user needs and system resource constraints.
- We introduce a novel Mixed Consistency model in order to meet the heterogeneity challenge by allowing both strong and weak replicas of the same object to coexist in the system at the same time.
- We propose an Access Constraints based approach to combine existing strong and weak consistency models together. Although we use sequential and causal consitency as an example in this thesis, this approach can be used to combine other consistency models together, for example, PRAM consistency.
- We introduce a new concept of responsiveness and propose a non-responsive process detector to tolerate crashed process, and slow process/communication links.
- We give the MC protocol which utilizes the existing implementations of strong and weak consistency models. The MC protocol only requires minimum modifications to the existing protocols.

- We implement the MC protocol on top of a publisher-subscriber based middleware system, Echo, which provides platform independent efficient transmission of large data. We conduct a series of experiments whose results meet our expectations.

8.2 Future Directions

Although we defined and provided implementation for MC, there are several interesting research issues that still need to be explored to meet the heterogeneity challenge of large information sharing systems. We briefly discuss some of the problems here:

- **Agile Dissemination Integration**. We currently implement our protocol on top of Echo, which provides efficient and platform-independent transmission of large data. It allowed us to focus on the MC protocol and understand how well the heterogeneity factors are addressed by our implementation. However, as a publisher-subscriber based middleware, addressing heterogeneity factors such as user needs and bandwidth constraints is not its main purpose. Our agile dissemination protocol, on the other hand, takes those factors into consideration when doing update dissemination. How to integrate our MC implementation with the agile dissemination protocol remains a research topic.

- **Multiple Consistency Models Support**. The current implementation of MC model supports SC and CC. We choose these two to demonstrate the MC model and our access constraints based approach. However, as we stated before, the access constraints based approach is not limited only to these two models. The MC model can be extended to cover other consistency models, such as PRAM (see Section 3.5). It will be an interesting research problem to explore and identify different consistency models that can be incorporated into MC.

- **Real Trace Driven Performance Experiments**. We currently conduct our experiments using sythetic traces. By adjusting the composition of the traces, we can expose the effect of different factors that impact the performance of the MC protocol. We do not consider other factors such as the burst of read and writer operations that

are not uncommon in real life systems. To run traces from real object sharing systems against our MC protocol will be an interesting task in the future.

- **More Comprehensive Performance Evaluation Platform**. We currently use CoC Warp cluster as our performance evaluation platform. The purpose is to expose the impact of different system parameters (and their combination) to the overall performance that the upper layer application experiences. We want to conduct these experiments in a controlled environment, i.e. cluster, where the results can be measured in a way such that other factors such as competing traffic, link failures along transmission path, variation of routing time, etc, can be largely ignored. We currently also conduct several tests with failure rates assumed. Future work should address those limitations by conducting performance evaluation experiments on top of a comprehensive platform against actual network traffic.

REFERENCES

[1] Adamec, J., Grf, M., Kleindienst, J., Plsil, F., and Turingma, P., "Supporting interoperability in corba via object services," Feb 26 2006.

[2] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P., "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[3] Agrawal, D., Choy, M., Leong, H. V., and Singh, A. K., "Mixed consistency: a model for parallel programming (extended abstract)," in *The 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 1994.

[4] Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S., "On implementing omega with weak reliability and synchrony assumptions," in *Proceedings of the twenty-second annual Symposium on Principles of Distributed Computing*, pp. 306–314, 2003.

[5] Ahamad, M., Bazzi, R., John, R., Kohli, P., and Neiger, G., "The power of processor consistency," in *5th ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[6] Ahamad, M., John, R., Kohli, P., and Neiger, G., "Causal memory: Meets the consistency and performance needs of distributed applications," in *ACM SIGOPS European Workshop*, pp. 45–50, 1994.

[7] Ahamad, M., Neiger, G., Burns, J. E., Hutto, P. W., and Kohli, P., "Causal memory: Definitions, implementation and programming," *Distributed Computing*, vol. 9, pp. 37–49, 1995.

[8] Ahamad, M. and Raynal, M., "Ordering vs timeliness: Two facets of consistency?," *Future Directions in Distributed Computing*, 2003.

[9] Alonso, R., Barbara, D., and Garcia-Molina, H., "Quasi-copies: Efficient data sharing for information retrieval systems," in *EDBT'88, LNCS 303*, 1988.

[10] Anderson, T., Dahlin, M., Neefe, J., Pat-terson, D., Roselli, D., and Wang, R., "Serverless network file systems," in *In Proceedings of the 15th Symposium on Operating System Principles. ACM*, (Copper Mountain Resort, Colorado), pp. 109–126, December 1995.

[11] Arora, A., Leon, G., and Wallace, S., "The corba replication service," As in Feb 26, 2006.

[12] Attiya, H. and Friedman, R., "A correctness condition for high-performance multiprocessors," in *Proc. of the 24th ACM Symposium on Theory of Computing (STOC)*, 1992.

[13] Attiya, H. and Welch, J. L., "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems*, 1994.

[14] Bakken, D., Zhan, Z., Jones, C., and Karr, D., "Middleware support for voting and data fusion," in *DSN '01: Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN'01)*, 2001.

[15] Barbara, D. and Garcia-Molina, H., "The case for controlled inconsistency in replicated data," in *the Workshop on the Managed of Replicated Data*, pp. 35–38, 1990.

[16] Birman, K. P., Hayden, M., Ozkasap, O., Xiao, Z., Budiu, M., and Minsky, Y., "Bimodal multicast," *ACM Transactions on Computer Systems*, vol. 17, no. 2, pp. 41–88, 1999.

[17] Bolosky, W. J., Douceur, J. R., Ely, D., and Theimer, M., "Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2000.

[18] Cao, P. and Liu, C., "Maintaining strong cache consistency in the world wide web," *IEEE Transactions on Computers*, vol. 47, no. 4, pp. 445–457, 1998.

[19] Carter, J., Bennett, J., and Zwaenepoel, W., "Implementation and performance of munin," in *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, 1991.

[20] Chandra, T. D. and Toueg, S., "Unreliable failure detectors for reliable distributed systems," in *Journal of the ACM (JACM)*, pp. 225–267, 1996.

[21] Chankhunthod, A., Danzig, P. B., Neerdaels, C., Schwartz, M. F., and Worrell, K. J., "A hierarchical internet object cache," in *USENIX Annual Technical Conference*, pp. 153–164, 1996.

[22] Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W., "Freenet: A distributed anonymous information storage and retrieval system," *Lecture Notes in Computer Science*, vol. 2009, pp. 46+, 2001.

[23] Clip2, "The gnutella protocol specification v0.4 document revision 1.2," *http://www9.limewire.com/developer/gnutella_protocol_0.4.pdf*.

[24] CORBA, "The corba homepage: http://www.corba.com," Dec 2 2006.

[25] Cox, A., de Lara, E., and Hu, W. Z. Y. C., "A performance comparison of homeless and home-based lazy release consistency protocols for software shared memory," in *Proc. of the 5th IEEE Symp. on High-Performance Computer Architecture (HPCA-5)*, 1999.

[26] Cukier, M., Ren, J., Sabnis, C., Henke, D., Pistole, J., Sanders, W., Bakken, D., Berman, M., Karr, D., and Schantz, R., "Aqua: An adaptive architecture that provides dependable distributed objects," in *Proceedings of the Seventeenth Symposium on Reliable Distributed Systems (SRDS-17)*, 1998.

[27] Datta, A., Hauswirth, M., and Aberer, K., "Updates in highly unreliable, replicated peer-to-peer systems," in *In the proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[28] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D., "Epidemic algorithms for replicated database maintenance," in *In the proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC)*, 1987.

[29] Douceur, J. R. and Wattenhofer, R. P., "Large-scale simulation of replica placement algorithms for a serverless distributed file system," in *9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2001.

[30] Drew, P. and Pu, C., "Asynchronous consistency restoration under epsilon serializability," *Technical Report OGI-CSE-93-004, Department of Computer Science and Engineering, Oregon Graduate Institute*, 1993.

[31] Eisenhauer, G., "The echo event delivery system, http://www-static.cc.gatech.edu/systems/projects/echo/eisenhauer02eed.pdf," Feb 26 2006.

[32] Eisenhauer, G. and Schwan, K., "An object-based infrastructure for program monitoring and steering," in *In Proceedings 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pp. 10–20, 1998.

[33] Eisenhauer, G., Schwan, K., and Bustamante, F. E., "Publish-subscribe for high-performance computing," in *IEEE Internet Computing 10(1): 40-47*, 2006.

[34] Emulab, "The emulab homepage: http://www.emulab.org," Dec 10 2006.

[35] Fernandez, A., Jimenez, E., and Cholvi, V., "On the interconnection of causal memory systems," in *Proc. of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, July 2000.

[36] Fischer, M., Lynch, N., and Paterson, M., "Impossibility of distributed consensus with one faulty process," in *Journal of ACM, Volume 32, Issue 2, Pages 374 - 382*, 1985.

[37] Gedik, B. and Liu, L., "Reliable peer-to-peer information monitoring through replication," in *SRDS 2003*.

[38] Gnutella, "The gnutella homepage: http://www.gnutella.com," Nov 12 2006.

[39] Goodman, J. R., "Cache consistency and sequential consistency," No. Technical Report 61, IEEE Scalable CoherenceInterface Working Group, 1989.

[40] Gwertzman, J. and Seltzer, M. I., "World wide web cache consistency," in *USENIX Annual Technical Conference*, pp. 141–152, 1996.

[41] Harvesf, C. and Blough, D. M., "The effect of replica placement on routing robustness in distributed hash tables," in *P2P '06: Proceedings of the Sixth IEEE International Conference on Peer-to-Peer Computing*, (Washington, DC, USA), pp. 57–6, IEEE Computer Society, 2006.

[42] Herlihy, M. P. and Wing, J. M., "Linearizability: A correctness condition for concurrent objects," *ACM TOPLAS*, vol. 12, pp. 463–492, July 1990.

[43] Jimenez, E., Fernandez, A., and Cholvi, V., "Decoupled interconnection of distributed memory models," in *Proc. of the 7th International Conference on Principles of Distributed Systems (OPODIS 2003)*, December 2003.

[44] Joseph, A. D., deLespinasse, A. F., Tauber, J. A., ord, D. K. G., and Kaashoek, M. F., "Rover: A toolkit for mobile information access," in *the 15th ACM Symposium on Operating Systems Principles*, pp. 156–171, 1995.

[45] Kangasharju, J., Ross, K., and Turner, D., "Secure and resilient peer-to-peer e-mail: Design and implementation," in *Proceedings of IEEE International Conference on P2P Computing, 2003.*, 2003.

[46] Kazza, "The kazza homepage: http://www.kazza.com," Nov 12 2006.

[47] Keleher, P., Cox, A. L., and Zwaenepoel, W., "Lazy release consistency for software distributed shared memory," in *Proc. of the 19th Annual Int'l Symp. on Computer Architecture (ISCA'92)*, pp. 13–21, 1992.

[48] Kong, L., Manohar, D. J., Ahamad, M., Subbiah, A., Sun, M., and Blough, D. M., "Agile store: Experience with quorum-based data replication techniques for adaptive byzantine fault tolerance," in *SRDS '05: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*, (Washington, DC, USA), pp. 143–154, IEEE Computer Society, 2005.

[49] Krishnakuma, N. and Bernstein, A., "Bounded ignorance in replicated systems," in *the ACM Principles of Database Systems*, 1991.

[50] Krishnaswamy, V., Ahamad, M., Raynal, M., and Bakken, D., "Shared state consistency for time-sensitive distributed applications," in *21th International Conference on Distributed Computing Systems (ICDCS)*, 2001.

[51] Ladin, R., Liscov, B., Shrira, L., and Ghemawat, S., "Lazy replication: Exploiting the semantics of distributed services," in *the Workshop on the Managed of Replicated Data*, pp. 31–34, 1990.

[52] Lamport, L., "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C28(9), pp. 690–691, 1979.

[53] Lamport, L., "On interprocess communication: Part i - basic basic formalism," *Distributed Computing*, pp. 1:77–85, 1986.

[54] Lamport, L., "On interprocess communication: Part ii - algorithms," *Distributed Computing*, pp. 1:86–101, 1986.

[55] Lipton, R. and Sandberg, J. S., "Pram: A scalable shared memory.," *Technical Report CS-TR-180-88, Princeton University, Dept. of Computer Science*, 1988.

[56] Liskov, B., Castro, M., Shrira, L., and Adya, A., "Providing persistent objects in distributed systems," *Lecture Notes in Computer Science*, vol. 1628, 1999.

[57] Liu, C. and Cao, P., "Maintaining strong cache consistency in the world-wide web," in *International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[58] Liu, L., "Maintaining Database consistency in the Presence of Schema Evolution," in *Proceedings of the Sixth IFIP TC-2 Working Conference on Data Semantics (DS-6)* (Meersman, R. and Mark, L., eds.), (Stone Mountain, Atlanta), Chapman Hall, London, 1995.

[59] McKinnon, D., Haugan, O., Damania, T., Dorow, K., Lawrence, W., and Bakken, D., "A configurable middleware framework with multiple quality of service properties for small embedded systems," in *Technical Report, WSU, TR-2002-37*, 2002.

[60] Muthitacharoen, A., Morris, R., Gil, T. M., and Chen, B., "Ivy: A read/write peer-to-peer file system," in *Proceedings of 5th Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[61] Napster, "The napster homepage: http://www.napster.com," Nov 12 2006.

[62] Narasimhan, P., Moser, L., and Melliar-Smith, P., "State synchronization and recovery for strongly consistent replicated corba objects," in *Proceedings of the IEEE Int. Conf. on Dependable Systems and Networks (DSN)*, 2001.

[63] Nelson, M. N., Khalidi, Y. A., and Madany, P. W., "The Spring file system," Tech. Rep. SMLI TR–93–10, 1993.

[64] Noe, J. D., Proudfoot, A. B., and Pu, C., "Replication in distributed systems: the eden experience," in *ACM '86: Proceedings of 1986 ACM Fall joint computer conference*, (Los Alamitos, CA, USA), pp. 1197–1209, IEEE Computer Society Press, 1986.

[65] OMG, "The omg homepage: http://www.omg.org," Feb 26 2006.

[66] Overview, E. H., "The emulab hardware overview homepage: http://www.emulab.net/docwrapper.php?docname=hardware.html," Dec 10 2006.

[67] Pu, C. and Leff, A., "Replica control in distributed systems: An asynchronous approach," in *SIGMOD Conference*, pp. 377–386, 1991.

[68] Raynal, M. and Schiper, A., "From causal consistency to sequential consistency in shared memory systems," in *15th Conference on Foundations of Software Technologies and Theoretical Computer Science*, (Bangalore, India), pp. 180–194, Springer-Verlag, 1995.

[69] Raynal, M. and Mizuno, M., "How to find his way in the jungle of consistency criteria for distributed objects memories (or how to escape from minos'labyrinth)," in *Proc. of the IEEE International Conference on Future Trends of Distributed Computing Systems*, September 1993.

[70] Rhea, S., Wells, C., Eaton, P., Geels, D., Zhao, B., Weatherspoon, H., and Kubiatowicz, J., "Maintenance-free global data storage," *IEEE Internet Computing 5(5), 40-49.*, 2001.

[71] Rowstron, A. I. T. and Druschel, P., "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility," in *Symposium on Operating Systems Principles (SOSP)*, pp. 188–201, 2001.

[72] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B., "Design and implementation of the Sun Network Filesystem," in *Proc. Summer 1985 USENIX Conf.*, (Portland OR (USA)), pp. 119–130, 1985.

[73] Satyanarayanan, M., Kistler, J. J., Kumar, P., Okasaki, M. E., Siegel, E. H., and Steere, D. C., "Coda: A highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 447–459, 1990.

[74] Singla, A., Ramachandran, U., and Hodgins, J. K., "Temporal notions of synchronization and consistency in beehive," in *ACM Symposium on Parallel Algorithms and Architectures*, pp. 211–220, 1997.

[75] Subbiah, A. and Blough, D. M., "An approach for fault tolerant and secure data storage in collaborative work environments," in *StorageSS '05: Proceedings of the 2005 ACM workshop on Storage security and survivability*, (New York, NY, USA), pp. 84–93, ACM Press, 2005.

[76] Terry, D. B., Theimer, M. M., Peterson, K., J.Demers, A., Spreitzer, M. J., and Hauser, C. H., "Managing update conflicts in bayou, a weakly connected replicated storage system," in *Proceedings of the 15th ACM SOSP*, 1995.

[77] Wolfson, O., Jajodia, S., and Huang, Y., "An adaptive data replication algorithm," *ACM Transactions on Database Systems*, vol. 22(4), pp. 255–314, 1997.

[78] Yu, B.-H., Huang, Z., Cranefield, S., and Purvis, M., "Homeless and home-based lazy release consistency protocols on distributed shared memory," in *27th Australasian Computer Science Conference (ACSC'04)*, 2004.

[79] Yu, H. and Vahdat, A., "Design and evaluation of a conit-based continuous consistency model for replicated services," in *ACM Transactions on Computer Systems (TOCS)*, 2002.

[80] Zhan, Z., Ahamad, M., and Raynal, M., "Mixed consistency model: Meeting data sharing needs of heterogeneous users," in *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, (Washington, DC, USA), pp. 209–218, IEEE Computer Society, 2005.

[81] Zhang, C. and Zhang, Z., "Trading replication consistency for performance and availability: an adaptive approach," in *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, (Washington, DC, USA), p. 687, IEEE Computer Society, 2003.

[82] Zhang, J., Liu, L., Pu, C., and Ammar, M., "Reliable peer-to-peer end system multicasting through replication," in *P2P '04: Proceedings of the Fourth International Conference on Peer-to-Peer Computing (P2P'04)*, (Washington, DC, USA), pp. 235–242, IEEE Computer Society, 2004.

[83] Zhou, Y., Iftode, L., and Li, K., "Performance evaluation of two home-based lazy release consistency protocols for shared memory virtual memory systems," in *Proc. of the 2nd Symp. on Operating Systems Design and Implementation (OSDI'96)*, pp. 75–88, 1996.